Capitalisation de la sûreté de fonctionnement des applications soumises à des adaptations dynamiques : le modèle exécutable Satin

Audrey Occello

UNIVERSITE DE NICE-SOPHIA ANTIPOLIS - UFR Sciences

ECOLE DOCTORALE STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H E S E

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice-Sophia Antipolis

Mention : Informatique

présentée et soutenue par

Audrey OCCELLO

CAPITALISATION DE LA SURETE DE FONCTIONNEMENT DES APPLICATIONS SOUMISES A DES ADAPTATIONS DYNAMIQUES : Le modèle exécutable Satin

Thèse dirigée par *Anne-Marie DERY-PINNA et Michel RIVEILL*

soutenue le *7 juin 2006*

Jury

Mme Laurence DUCHIEN	Professeur des Universités	Présidente
M. Didier BERT	Chargé de Recherche CNRS	Rapporteur
M. Jean BEZIVIN	Professeur des Universités	Rapporteur
M. Gunther KNIESEL	Senior Lecturer	Rapporteur
M. Philippe COLLET	Maître de Conférences	Examinateur
Mme Anne-Marie DERY-PINNA	Maître de Conférences	Directrice de thèse
M. Michel RIVEILL	Professeur des Universités	Directeur de thèse

Remerciements

Je tiens tout d'abord à remercier Anne-Marie Dery-Pinna et Mireille Blay-Fornarino pour leur accueil spontané, au sein de l'équipe RAINBOW du laboratoire I3S, il y déjà six ans, pour mon projet de fin d'étude d'ingénieur. Elles m'ont donné la motivation pour entamer une année de DEA et pour poursuivre dans la recherche.

Je remercie chaleureusement Anne-Marie Dery-Pinna et Michel Riveill d'avoir accepté d'être mes directeurs de thèse. Merci à vous deux et à Mireille d'avoir relu la thèse. Combien de fois avez-vous regretté votre choix depuis les maintes relectures du manuscrit ! Heureusement que vous avez tenu bon car il ne serait pas ce qu'il est sans vous.

Je souhaite remercier à nouveau Anne-Marie de m'avoir supporté et d'avoir toujours su me donner les bons conseils même dans les moments de doute, où je suis difficile à convaincre ... Travailler avec toi a été d'une facilité immense : nous avons peut être inventé la transmission de pensées (une autre forme d'architecture répartie !).

Je remercie Jean Bezivin, Didier Bert et Gunther Kniesel pour m'avoir fait le plaisir de rapporter ma thèse. Leurs remarques m'ont permis d'améliorer la qualité du manuscrit. Je remercie tout particulièrement Gunther qui a bravé les barrières de la langue pour évaluer mon travail. Je remercie encore Didier pour le temps qu'il a passé à m'initier à B !

Je remercie également Laurence Duchien et Philippe Collet d'avoir accepté de participer à mon Jury. Merci aussi pour toutes nos discussions !

Je tiens à remercier l'ESSI pour son accueil et pour l'aide au quotidien de ses enseignants et de son personnel. Je pense notamment à Hélène, Dominique, Jean-Louis, Muriel, ...

Merci à tous les membres de l'équipe RAINBOW pour tous les moments de détente passés autour de nos fameux p'tits dej' et goûters.

Merci aux membres de ma famille pour leur patience pendant ces derniers mois difficiles. Je tiens à remercier mes parents, qui m'ont donné l'envie d'apprendre et la valeur du travail. Enfin (last but not least), merci à maman pour son soutien inconditionnel et permanent : je te dédie cette thèse.

Table des matières

Table des figures

Liste des tableaux

« Le génie est fait d'un dixième d'inspiration et de neuf dixièmes de transpiration »

Thomas Edison

« Les manuscrits raturés, barbouillés, mêlés, indéchiffrables attestent la peine qu'ils m'ont coûtée »

Jean Jacques Rousseau

1 Introduction

Contexte et enjeux

La capacité d'adaptation à l'exécution représente une exigence importante des logiciels modernes. A certains moments du cycle de vie, les approches statiques d'adaptation ne peuvent, en effet, pas être utilisées car l'interruption complète de l'application ne peut pas être tolérée. A l'âge de l'Informatique Ubiquitaire, un très grand nombre d'applications est confronté à un environnement communicant évolutif et à un comportement imprévisible des utilisateurs. Pour ces applications dites sensibles au contexte, la dynamicité se justifie par la fréquence de l'adaptation. D'autres types d'applications nécessitent également d'être adaptées dynamiquement. Les applications dites à haute disponibilité, comme les serveurs d'application (commerces électroniques ou systèmes bancaires) accessibles via Internet ne peuvent tolérer une rupture de service. Dans de tels cas, il est nécessaire d'adapter dynamiquement ce type de systèmes pour en minimiser la perturbation et garantir une qualité de service acceptable. De ce fait, les approches à composants et à aspects prennent de plus en plus souvent en charge les adaptations dynamiques (ajout / retrait de fonctionnalités, modification du comportement des fonctionnalités, modification des assemblages de composants, ...) permettant ainsi une évolution de plus en plus aisée des applications.

Alors que les technologies permettant l'adaptation dynamique arrivent à maturité et que nous en maîtrisons les mécanismes, il devient essentiel de considérer la validité de tout système adaptatif. En effet, si l'on considère que la sûreté de fonctionnement d'une application est la propriété permettant aux utilisateurs d'un système de placer une confiance justifiée dans le service qu'il leur délivre [56] alors il faut garantir lors d'une adaptation que cette propriété est préservée. Autrement dit, une adaptation n'est pas « sûre » à partir du moment où le service que fournit l'application après adaptation diverge du service attendu par l'utilisateur. Un certain nombre de techniques (typage, model-checking, ...) ont vu le jour pour fournir des méthodes et des outils afin de concevoir et implémenter les systèmes informatiques de façon sûre. Or, les adaptations peuvent introduire des modifications qui vont compromettre la sûreté de fonctionnement des applications qui a été établie initialement (introduction d'un appel à une méthode inconnue, ajout de comportements contradictoires ou aléatoires, suppression d'un composant requis ou ajout d'un composant de type incorrect dans un assemblage, introduction de cycles, ...).

Trois points différencient la problématique de la sûreté en environnement statique de celle en environnement dynamique. Premièrement, une application statique peut utiliser les techniques sus-citées pour rétablir son bon fonctionnement. A l'inverse, les applications adaptées dynamiquement ne peuvent pas bénéficier directement de telles techniques qui ne sont pas faites pour traiter des problèmes de sûreté dans un contexte dynamique. Deuxièmement, une adaptation statique peut être mise en œuvre à n'importe moment contrairement à une adaptation dynamique qui risque de se produire à un moment inadéquat dans l'exécution de l'application. Troisièmement, si une adaptation statique introduit un problème de sûreté, il est relativement aisé de revenir en arrière et de corriger le problème. Par contre, prendre en compte une adaptation dynamiquement implique de prévoir de traiter les problèmes de sûreté. C'est à dire qu'il faut savoir les détecter en cours d'exécution, et ce de préférence avant qu'ils ne se produisent, et les résoudre parallèlement à l'exécution de l'application sans perturber les utilisateurs.

Suivant les applications ciblées, la dynamicité d'une adaptation perd son sens sans cette garantie en terme de sûreté. En particulier, les applications dites critiques, comme les services de télécommunication, l'utilisation d'outils permettant les adaptations dynamiques n'est pas encore acceptable de par les problèmes de sûreté qu'elle entraine. Actuellement, il n'existe pas de solution consensuelle et appropriée au problème de la sûreté des adaptations dynamiques.

Contributions de la thèse

Nous proposons de fiabiliser le processus d'adaptation dynamique indépendamment de la plate-forme cible, d'identifier la sûreté d'une adaptation, et de déterminer quand et comment les modifications liées à une adaptation peuvent être prises en compte et mises en œuvre de façon sûre dans l'exécution de l'application.

Cette approche est basée sur un modèle nommé Satin, décrit en UML [91], sur lequel sont exprimées des propriétés de sûreté à l'aide du langage de contraintes OCL [120]. Le modèle Satin est mis en œuvre sous la forme d'un service de sûreté que les plates-formes à composants et les canevas orientés aspects peuvent interroger pour déterminer si une adaptation donnée risque de briser la sûreté de fonctionnement de l'application.

Un prototype de ce service a été développé en Java et a été testé avec les plates-formes Noah [17] et Fractal [86] afin de valider l'applicabilité de l'approche proposée. Enfin, l'adéquation du degré d'abstraction et d'extensibilité du modèle est évalué en prenant en compte les adaptations liées à la mobilité des composants.

Les contributions de cette thèse sont les suivantes :

- Une étude des systèmes adaptatifs dynamiques et de la façon dont ces systèmes gèrent les problèmes de sûreté dus aux adaptations ;
- Une analyse de la notion d'adaptation dynamique et la mise en évidence d'une liste de propriétés de sûreté ;
- Une démarche extensible basée sur la définition d'un modèle de sûreté d'adaptation, Satin, indépendant des plates-formes technologiques et décoré de contraintes formalisant les propriétés de sûreté ;
- Une méthodologie visant à prouver formellement le modèle de sûreté d'adaptation ;
- Une approche basée sur la mise en œuvre d'un service de sûreté pour rendre opérationnel le modèle Satin et qui peut être réutilisée pour concrétiser d'autres modèles exécutables.

Plan de la thèse

Ce document est composé de trois parties. La **première partie** dresse un état de l'art en matière de systèmes adaptatifs dynamiques et de sûreté de fonctionnement des applications. Elle présente les différentes formes que peut prendre une adaptation et les classes d'erreurs qu'elle peut engendrer. Elle comporte trois chapitres :

- Le chapitre 2 présente différents travaux autour des composants et des aspects permettant l'adaptation dynamique. Chaque approche est illustrée par un exemple.
- Le chapitre 3 décrit les types de problèmes de sûreté rencontrés dans les systèmes adaptatifs dynamiques et comment les différents travaux décrits dans le chapitre 2 y répondent.
- Le chapitre 4 conclut la première partie en synthétisant l'étude qui y a été faite et affine les objectifs de cette thèse.

La **deuxième partie** présente le cœur de la contribution de la thèse au domaine de la sûreté de fonctionnement en présence d'adaptations dynamiques. Elle est constituée de trois chapitres :

- Le chapitre 5 présente l'approche suivie, une illustration de sa mise en œuvre à travers le modèle Satin et son utilisation sur un exemple.
- Le chapitre 6 détaille les deux éléments centraux du modèle Satin : une forme de typage dédiée à la prise en compte des adaptations et une description des adaptations indépendante des systèmes adaptatifs dynamiques.
- Le chapitre 7 décrit formellement le modèle Satin et sa validation.

La **troisième partie** vise à démontrer l'adéquation du degré d'abstraction du modèle proposé en analysant ses possibilités en termes d'applicabilité et d'extensibilité. Elle est divisée en 2 chapitres :

- Le chapitre 8 expose l'applicabilité du modèle en étudiant deux techniques de projection : la projection par transformation du modèle vers différentes plates-formes technologiques et la conception d'un service de sûreté concrétisant le modèle Satin.
- Le chapitre 9 présente une extension du modèle pour gérer l'adaptabilité à la mobilité et montre l'impact de cette extension sur le modèle.

Un dernier chapitre regroupe les conclusions et perspectives de ce travail. Une liste des publications produites pendant cette thèse, ainsi qu'une liste des acronymes utilisés sont disponibles en annexes E et F.

Première partie

Etat de l’art

« On ne peut rien enseigner à autrui, on ne peut que l'aider à le découvrir en lui-même »

Galilée

2

Domaine d'étude : Systèmes adaptatifs dynamiques

AFIN de motiver différents besoins en terme d'adaptation dynamique des applications, nous utilisons, comme exemple fil rouge, une application de gestion d'agendas. Nous disposons d'entités logicielles[1] implémentant les fonctionnalités[2] de base d'un agenda (ajout, suppression et consultation de rendez-vous).

Une utilisation d'une telle entité logicielle dans le cadre d'agendas collaboratifs peut se faire par adaptation de l'existant pour permettre à certains utilisateurs de conditionner l'ajout de rendez-vous selon l'appelant par exemple ou de notifier leurs indisponibilités à d'autres agendas. Dans ce cas, le besoin en terme d'adaptation est de modifier le comportement[3] de l'ajout de rendez-vous pour l'agenda à adapter.

Si l'utilisateur consulte son agenda sur un assistant personnel (PDA, Personal Digital Assistant), il peut être amené à visualiser ou mémoriser ses rendez-vous différemment selon les ressources disponibles. Pour économiser la batterie par exemple, l'utilisateur peut vouloir déconnecter son agenda de la base de données (à laquelle il est relié lorsque son PDA est suffisamment chargé) et le connecter à un système de fichiers pour effectuer les sauvegardes. Pour libérer de la mémoire lorsqu'il utilise plusieurs applications simultanément, l'utilisateur peut décider de supprimer temporairement la visualisation graphique de l'agenda. Dans ce cas, le besoin en terme d'adaptation est de faire évoluer les entités utilisées par l'agenda et donc de modifier les liens entre les entités logicielles formant un réseau d'entités coopérantes.

Enfin, pour acquérir de nouveaux services dynamiquement, afin de limiter l'accès à un agenda par exemple, de nouveaux paramètres (mot de passe, adresse d'une machine, ...) sont nécessaires. Dans ce cas, le besoin en terme d'adaptation est la modification de certaines signatures de fonctionnalités avec comme conséquence une modification des interfaces des agendas en question[4].

[1] Afin de ne s'attacher à aucun système d'étude particulier, nous choisissons volontairement le terme d'entité logicielle, pour désigner les objets, les agents, les composants, les services Web, etc

[2] L'ensemble des méthodes implémentées par une entité logicielle. Dans la suite, nous utilisons indistinctement les termes fonctionnalités et méthodes.

[3] Par le terme « comportement », nous désignons l'exécution du code d'une méthode.

[4] Même si ces extensions d'interface sont rendues transparentes vis-à-vis de l'utilisateur, elles restent cependant nécessaires à la gestion des services.

De ces exemples, nous dégageons sept types d'adaptation élémentaires catégorisés en trois familles : 1) l'ajout, le retrait de fonctionnalités et la modification de la signature des fonctionnalités qui conduisent à l'**évolution des interfaces** des entités logicielles, 2) la modification répétée du comportement associé à une fonctionnalité qui conduit à la **composition comportementale** des nouveaux comportements avec le comportement préexistant, et 3) l'ajout, le retrait et le remplacement d'entités logicielles qui conduisent à l'**altération des assemblages**. Ces types d'adaptation peuvent bien évidemment être combinés pour obtenir des adaptations plus complexes.

Dans la suite de ce chapitre nous présentons les deux principaux courants permettant l'adaptation dynamique : les approches orientées aspects 2.1 et les approches à base de composants 2.2. Nous nous basons sur les familles d'adaptations élémentaires pour classer chaque approche.

2.1 Approches orientées aspects

La séparation des préoccupations (SoC, Separation of Concern) est un domaine important du génie logiciel. Modulariser, hériter, déléguer sont des techniques « orientées objet » pour adapter, réutiliser, évoluer ... Cependant, le paradigme objet n'élimine pas toujours l'entrelacement du code fonctionnel[5] et du code non fonctionnel[6] à l'intérieur d'une classe et l'éparpillement du code non fonctionnel à travers les classes. De ce fait, la réutilisation des objets demande des transformations importantes et répétitives lorsqu'ils sont sortis du cadre de leur utilisation première.

La programmation dite par aspects (AOP pour Aspect Oriented Programming) [19] vise à permettre une séparation des différentes préoccupations des programmeurs face à la réutilisation d'un programme. Le but de l'AOP est de supprimer les dépendances entre code technique (i.e. non fonctionnel) et code métier (i.e. fonctionnel) pour isoler de façon encore plus poussée les modules. Ainsi l'AOP permet une nouvelle forme de réutilisation à travers une séparation des différentes préoccupations des programmeurs. Un jeu d'aspects vient se greffer sur le programme de base pour en modifier le comportement. L'AOP peut être mise en œuvre aussi bien statiquement que dynamiquement.

Nous intéressant à l'adaptation dynamique, nous écartons ainsi de cette étude les approches statiques telles que AspectJ [59]. Cependant, AspectJ étant le langage de référence et le précurseur de l'AOP, nous reprenons le même vocabulaire et l'utilisons comme point de référence pour introduire les concepts de base communément utilisés dans les différents langages d'aspects dans la section 2.1.1. Les sections suivantes présentent quelques canevas orientés aspects permettant d'adapter les applications au cours de leur exécution.

2.1.1 Concepts de base

Aspect - Un aspect est une abstraction. Il correspond à la définition de structures et de comportements qui se superposent à la définition de l'application métier. Les *aspects* correspondent aux propriétés qui « coupent » l'application métier.

Point de jointure - Un point de jointure est un point du flot d'exécution de l'application qui peut être contrôlé et où les aspects peuvent intervenir (invocation de méthode, accès à un slot, initialisation, ...).

Coupe - Une coupe est une combinaison de valeurs de points de jointure (nom de classe, de méthode, ...). La définition d'une coupe permet donc de désigner les points de jointure à utiliser pour « interfacer » chaque aspect avec l'application.

Advice - Un advice est une structure d'aspect permettant de décrire les actions à effectuer lorsqu'une coupe est identifiée.

[5] Les fonctionnalités offertes par une application.

[6] Le code de contrôle de la stratégie d'implantation : persistance, sécurité, transactions, ...

Tissage - La dernière étape de la construction d'une application à partir d'aspects consiste à assembler les différents « modules » constituant la dite application. Cette étape, nommée tissage, implique l'utilisation d'un mécanisme de composition (implicite ou explicite) lorsque plusieurs aspects utilisent les mêmes points de jonction permettant ainsi leur superposition.

Dans la suite, nous nous intéressons à trois approches orientées aspects permettant le tissage dynamique d'aspects (ajout et retrait dynamique d'aspects). JAC [97], Composition Filters [13] et Noah [17] ont été choisis pour leurs différences en terme d'expression des aspects et de gestion de la composition d'aspects.

2.1.2 JAC

JAC (Java Aspect Components) [97] est un framework écrit en Java dédié au développement d'applications orientées aspects. Il met en œuvre l'API (Application Programming Interface) AOP Alliance [7] (initiative pour un consensus autour des frameworks orientés aspects) et prend en charge la composition d'aspects statiquement aussi bien que dynamiquement. Le modèle de programmation de JAC est composé principalement de 5 grandes parties :

1. Un programme de base constitué d'un ensemble d'objets écrits en Java standard.
2. Les composants d'aspect (*aspect components*) permettent de modifier le programme de base. Ils sont définis par une coupe et sont associés à des wrappeurs.
3. Les wrappeurs (advices) sont de 3 types :
   - une méthode « enveloppante » (*wrapping method*) permet de capturer l'appel à une méthode et d'effectuer un traitement avant ou après ou à la place de l'exécution réelle de la méthode,
   - une méthode « rôle » (*role method*) permet d'étendre les fonctionnalités d'un objet,
   - un gestionnaire d'exceptions (*exception handler*) traite les cas d'erreur non prévus dans le programme de base.
4. L'aspect de composition (*Composition aspect*) est un aspect spécial utilisé par le framework pour gérer les problèmes de composition d'aspects. Cet aspect définit l'ordre d'enchaînement des wrappeurs et exprime les dépendances ou conflits entre wrappeurs.
5. Le *weaver* a la charge de tisser les composants d'aspect sur un objet de base.

L'aspect de composition permet de définir des politiques de composition « sautant » l'exécution de certains aspects en fonction du contexte, de règles de dépendances et d'incompatibilités inter-aspects. Ainsi, lors de l'invocation d'une méthode, l'appel est intercepté et les aspects qui lui sont rattachés sont exécutés ou non. Les composants d'aspect peuvent être ajoutés, supprimés et réordonnancés au cours de l'exécution d'une application.

Les wrappeurs permettent de réaliser deux types d'adaptations élémentaires explicitement et dynamiquement : l'ajout de nouvelles fonctionnalités à un objet appartenant à la famille **évolution des interfaces** et la modification du comportement d'une fonctionnalité existante d'un objet appartenant à la famille **composition comportementale**. Revenons à l'exemple de l'agenda décrit dans l'introduction de ce chapitre. Nos agendas sont des objets java dont la classe `AgendaImpl` est décrite ci-dessous.

```
Class DiaryImpl {
   protected Hashtable rdvs;
   ...
   public void addRdv(Rdv r) { rdvs.put(r.getDate(), r);
   public void removeRdv(Rdv r) { rdvs.remove(r.getDate()); }
   public Rdv getRdv(Date d) { rdvs.get(d); }
   ...
}
```

Supposons que l'on veuille tracer les appels à l'accesseur `getRdv` des agendas. Dans ce cas, on définit un aspect de trace `TraceAC` (cf la figure 2.1). La coupe de l'advice correspond à tracer les appels à `getRdv` de n'importe quel objet de type `AgendaImpl` sur lequel on tisse `TraceAC`.

```
import org.objectweb.jac.core.AspectComponent;
public class TraceAC extends AspectComponent {
   public TraceAC() {
     pointcut( // definition d'une coupe
       ".*", // // instances ciblées
       "AgendaImpl", // classe
       "getRdv(Date):Rdv", // methode contrôlée
       "TraceWrapper", // wrappeur
       null, false
     );
   }
}
```

FIG. 2.1 – Exemple de définition d'un composant d'aspect de trace avec JAC

Le comportement du composant d'aspect est spécifié par le wrappeur qui lui est associé, en l'occurrence `TraceWrapper` (cf figure 2.2). Ce wrappeur comporte principalement trois actions : 1) affiche le nom et les arguments de la méthode interceptée (lignes 8-9), 2) passe la main au prochain wrappeur ou bien exécute la méthode (ligne 10), 3) affiche le résultat de l'étape 2 (ligne 12). Ce wrappeur correspond donc à une adaptation de type **modification du comportement d'une fonctionnalité**.

```
1  import org.aopalliance.intercept.MethodInvocation;
2  import org.objectweb.jac.core.Wrapper;
3  public class TraceWrapper extends Wrapper {
4    private Display display;
5    public CheckingWrapper(AspectComponent ac) { super(ac); }
6    public Object invoke(MethodInvocation mi ) {
7      Object ret = null;
8      System.out.println("<< calling "+mi.getMethod().getName()+
9                         " with "+mi.getArguments()+">>");
10     ret = proceed(mi); // execution du prochain wrappeur
11                        // ou de la méthode initiale
12     System.out.println("<< "+mi.getMethod().getName()+" returned "+ret+">>");
13     return ret;
14   }
15 }
```

FIG. 2.2 – Wrappeur associé au composant d'aspect de trace

Pour plus de généricité et afin d'utiliser les aspects pour différentes classes d'objets, le processus de tissage peut également être configuré par un fichier de propriété (au lieu d'être spécifié en « dur » dans le code de l'aspect) paramétrant l'endroit (classe, objet, méthode) et le moment (après la création de *n* instances par exemple) où les wrappeurs seront tissés. La définition de l'aspect est alors légèrement différente comme le montre la figure 2.3. Par exemple, le fichier de configuration du tissage comportant la ligne suivante permet de tracer les objets de n'importe quelle classe comportant au moins un accesseur en lecture.

```
addTrace ALL ALL "FIELDGETTERS";
```

```
import org.objectweb.jac.core.AspectComponent;
public class TraceAC extends AspectComponent {
    public TraceAC() {}
    public void addTrace(String objects, String classes, String methods) {
        pointcut(objects, classes, methods, "TraceWrapper", null, false);
    }
}
```

FIG. 2.3 – Exemple de définition d'un composant d'aspect de trace générique avec JAC

2.1.3 Filtres de composition

Le modèle des filtres de composition [13] est une extension modulaire du modèle orienté objet dont la notion centrale est le *filtre* permettant de changer la structure et le comportement des objets par interception des messages.

Les filtres sont groupés en unités appelées *modules* de filtres (les aspects). Ces modules sont l'unité d'instanciation et de réutilisation des filtres. Les modules sont composés de deux parties : une spécification de filtres (indépendante des langages de programmation) et une partie implémentation (dépendante d'un langage de programmation tels que Java, C++ et Smalltalk).

Les objets de l'application comportent deux collections ordonnées de filtres, l'une est destinée au traitement des messages reçus et l'autre est destinée au traitement des messages envoyés. Tous les messages entrant et sortant sont interceptés et soumis aux filtres avant d'être éventuellement traités par l'objet lui-même. Le mécanisme de tissage permettant de composer les filtres avec l'application est appelé la *superimposition*. La spécification d'une superimposition décrit le lieu dans l'application où les filtres vont être ajoutés.

Un filtre est composé d'éléments de filtre. Lorsqu'un message est reçu ou envoyé, il est intercepté successivement par chaque élément de filtre jusqu'à ce que l'un d'eux l'accepte ou que tous le rejettent. Chaque élément de filtre commence par demander si l'objet superimposé est dans un état donné (grâce aux méthodes *conditions*) puis si l'objet courant est dans le bon état, on vérifie que le message correspond à la *forme* de l'élément de filtre. L'ensemble des formes des éléments d'un filtre correspond donc à une coupe d'aspect et le type du filtre correspond à un advice d'aspect.

Il existe différents types de filtres prédéfinis dans le modèle. L'action réalisée lors de l'acceptation ou le rejet d'un message dépend de ce type. Par exemple, pour les filtres de type *Error*, l'acceptation d'un message se traduit par la levée d'une exception alors que son rejet conduit à sa transmission au filtre suivant. Pour les filtres du type *Wait*, l'acceptation d'un message conduit à sa transmission au filtre suivant, alors que son rejet conduit à sa suspension jusqu'à ce qu'il puisse être accepté. En cas d'acceptation, les filtres de type *Substitution* permettent de changer la cible, le nom ou les arguments du message intercepté, les filtres de type *Dispatch* de déléguer le message à l'objet lui-même ou tout autre objet et les filtres de type *Meta* de réifier le message intercepté et de le transmettre en paramètre d'un autre message à un objet[7]. En cas de rejet, les filtres des trois types précédents conduisent à sa transmission au filtre suivant.

Il existe plusieurs implémentations du modèle des filtres de composition. Certaines implémentations sont compilées et ne proposent donc que des adaptations statiques telles que ComposeJ et ConcernJ (au dessus de Java) [122, 107]. D'autres sont interprétées et offrent donc des possibilités en terme d'adaptations dynamiques telles que Sina (au dessus de Smalltalk) [61] et Compose* (au dessus de .Net) [42]. Dans la suite, nous nous focalisons sur Compose*.

Les filtres de composition permettent donc de réaliser trois types d'adaptations élémentaires explicitement et dynamiquement : l'ajout (via le filtre de type Substitution, Dispatch ou Meta) et

[7] L'objet qui reçoit le message peut manipuler le message réifié et éventuellement réactiver son exécution.

le retrait (via le filtre de type Error) de fonctionnalités à un objet appartenant à la famille **évolution des interfaces** et la modification du comportement d'une fonctionnalité existante d'un objet (via une combinaison quelconque de filtres de tous types) appartenant à la famille **composition comportementale**.

La spécification de la figure 2.4 définit le module de filtres `Trace`. Compose* permet de décrire les modules directement en manipulant un MOP (Meta Object Protocol) opérationnel. Cependant, pour plus de lisibilité, nous utilisons le langage de spécification du modèle. Le module `Trace` définit un filtre de type Meta, qui intercepte tous les messages et les envoie sous forme réifiée en tant qu'argument au message `affiche` de l'objet `afficheur`. Le module défini également un filtre de type Dispatch qui permet d'assurer que les méthodes `logOn()` et `logOff()` puissent être appelées sur l'objet superimposé (les appels seront redirigés vers l'instance du module `Trace` associée à l'objet superimposé). Une instance de ce module permet donc de modifier le comportement de l'ensemble des messages de l'objet qu'elle surimpose et permet à l'objet superimposé de traiter les deux méthodes (`logOn()` et `logOff()`) comme si elles faisaient partie de son interface.

```
filtermodule Trace begin
  externals afficheur : Afficheur; // référence à une instance partagée
  internals logOn : boolean; // créé au moment de la superimposition
  methods logOn(); logOff(); // méthodes définie pour les filtres du module
  conditions loggingEnabled; // méthodes sans effet de bord

  inputfilters  // deux fitres en entrée
    trace : Meta = { loggingEnabled=>[*]afficheur.affiche };
    nontraces : Dispatch = { loggingOn, loggingOff };

  implementation in Java;  // par exemple
    class TraceFilterModule {
      boolean logOn;
      boolean loggingEnabled() { return logOn; };
      void logOn() { logOn = true; };
      void logOff() { logOn = flase; };
  end implementation;
end filtermodule Trace;
```

FIG. 2.4 – Exemple de module de filtres de composition pour la trace

Ensuite, il est possible de superimposer le module de trace aux objets de classe Agenda. La superimposition est spécifiée de la manière suivante :

```
superimposition begin
  selectors  // selectionne les objets instances directes de la classe Agenda
    superimposedObjects = { *->accept(oclIsTypeOf(Agenda)) };
  methods  // association des methodes du module à une implémentation
    superimposedObjects <- { TraceFilterModule.loggingOn(), TraceFilterModule.loggingOff() };
  conditions  // association des méthodes sans effet de bord du module à une implémentation
    superimposedObjects <- TraceFilterModule.loggingEnabled;
  filtermodules  // superimpose les filtres du module Trace à tous les agendas
   superimposedObjects <- Trace;
end superimposition;
```

Avec Compose*, la superimposition peut également être directement programmée en utilisant le MOP.

```
TraceFilterModule trace = new TraceFilterModule();
ObjectManager.getObjectManagerFor(agendaDeAnneMarie).addFilterModule(trace);
agendaDeAnneMarie.logOn();
```

2.1.4 Noah

Noah [17] est un framework fournissant une couche d'adaptation dynamique pouvant être positionnée au dessus de différents langages (Java, C++) ou plates-formes telles que EJB (Enterprise JavaBeans) [104], .Net [79]. Le framework permet de modifier le comportement des entités logicielles de manière déclarative en utilisant le langage d'aspect ISL (Interaction Specification Language) [12], indépendant de tout langage d'implémentation.

Le framework est basé sur l'expression des aspects comme des entités de premières classes avec les propriétés suivantes :

- Un schéma d'interactions (i.e. un aspect), dé est composé de règles d'interactions, des règles de réécriture spécifiant les modifications comportementales d'entités logicielles,
- Une règle d'interaction est composée de deux parties : une spécifiant l'endroit où la règle doit être tissée (i.e. la coupe) et une autre décrivant la modification comportementale souhaitée (i.e. l'advice),
- Les interactions sont les instances de schémas et représentent le tissage d'un schéma à un ensemble donné d'entités logicielles,
- L'interface d'une entité logicielle à laquelle on a tissé un ou plusieurs schémas d'interactions n'est pas modifiée,
- La gestion du tissage des schémas est basée sur une composition des règles d'interactions qui respecte les propriétés de commutativité et d'associativité.

Le framework définit une API accessible depuis un serveur d'interactions qui permet de :

- définir dynamiquement de nouveaux schémas d'interactions (qui constituent ainsi une bibliothèque au niveau du serveur) à partir d'ISL,
- de tisser et de dé-tisser les schémas sur des entités logicielles par la création et la destruction d'interactions,
- d'introspecter les interactions pour connaître les entités logicielles adaptées et les schémas qui leur ont été tissés.

Le tissage de schémas d'interactions permet donc de mettre en œuvre explicitement et dynamiquement des adaptations élémentaires de la famille **composition comportementale**. Le schéma d'interactions `trace` défini ci-après permet de tracer l'exécution d'entités logicielles. Ce schéma prend en paramètre `obj` et `display` représentant les entités logicielles sur lesquelles ce schéma peut être tissé et comporte une seule règle d'interactions. L'utilisation de l'expression « * » dans la coupe de cette règle (avant l'opérateur de réécriture « -> ») permet de désigner de façon générique toutes les méthodes de l'objet auquel on tisse la règle.

```
Interaction trace(obj, display) {
   obj.* -> obj._call(); display.print(_call)
}
```

Reprenons l'exemple des agendas. Si le schéma d'interactions `trace` est tissé sur une entité logicielle *agendaDeMichel* (en tant que `obj`) et sur une entité logicielle *afficheur1* (en tant que `display`), par exemple, le comportement de ces deux entités est modifié comme suit : à chaque ajout, retrait ou quelconque autre exécution de méthodes fournies par *agendaDeMichel*, *afficheur1* affiche sur la sortie standard la méthode en cours d'exécution.

A travers les trois exemples d'approches orientées aspects, nous pouvons constater que la mise en œuvre des adaptations sous forme d'aspects permet de modifier des entités de type différent.

En effet, les wrappeurs de JAC [97], les filtres de composition [13] ainsi que les règles d'interactions de Noah [17] peuvent modifier toute entité logicielle satisfaisant la coupe définie par l'aspect. Dans le prochain chapitre, nous allons examiner un autre mécanisme de mise en œuvre des adaptations : l'approche à composants.

2.2 Approches par composants

Les composants logiciels [50, 113] constituent aujourd'hui une technologie phare de l'industrie du logiciel. Une raison à cette situation est la volonté de beaucoup d'industriels de capitaliser leur code et de fournir des logiciels sur étagère (COTS, Component Off The Shelf) pour accroître leur productivité et réduire le temps entre l'identification d'un besoin et la fourniture d'une solution. Aussi, la programmation par composants introduit-elle la notion d'assemblage comme technique de réutilisation des composants. La plupart des plates-formes à composants facilitent l'évolution statique ou dynamique d'une application construite à partir d'assemblages de composants par l'ajout, le retrait ou le remplacement de composants.

La norme ISO (International Standardization Organization) RM-ODP (Open Distributed Processing Reference Model) [55, 54, 53, 52] définit un ensemble de concepts dont les grandes lignes sont repris dans la plupart des modèles à composants. Ces concepts sont explicités en section 2.2.1. Les sections suivantes présentent quelques modèles et plates-formes à composants permettant d'adapter les applications au cours de leur exécution.

2.2.1 Concepts de base

Type de composant - Un type de composant est la définition abstraite d'une entité logicielle (de la même façon que la classe l'est pour un objet). Il est caractérisé par des interfaces. Il est possible de fournir plusieurs implantations pour un même type de composant et une implantation de composant peut être substituée par une autre implantation d'un type compatible. Il existe deux sortes d'interfaces, indispensables pour réaliser des assemblages :

- les *interfaces fournies* (ou d'entrée) par le composant sont du même ordre que les interfaces des objets : elles définissent les fonctionnalités offertes par un type de composant donné, en énumérant les signatures de ces dernières ;
- les *interfaces requises* (ou de sortie) par le type de composant représentent une évolution par rapport aux interfaces classiques de l'approche objet. Dans le cas des objets, une référence sur un objet utilisé est enfouie au cœur du code. Dans le cas des composants, les interfaces des types de composants utilisés sont exprimées au niveau du type de composant. Il est fortement recommandé d'exploiter cet enrichissement des types qui tend à expliciter les dépendances entre composants afin de faciliter la substitution de composants, d'une part, et la gestion des connexions entre des types de composants d'autre part.

Les interfaces définissent les points de communication des composants et sont utilisées pour leur invocation et leur interconnexion. En effet, il n'est pas possible d'utiliser un composant sans référencer une de ses interfaces d'entrée. Un mode de communication est spécifié pour chaque interface requise afin de préciser le mode d'interaction avec le type de composant qui la définit (mode synchrone, mode asynchrone, synchronisation, ...).

Implantation de composant - L'implantation d'un type de composant regroupe deux notions : l'implantation fonctionnelle et l'implantation non fonctionnelle. L'implantation fonctionnelle d'un composant représente sa mise en œuvre d'un point de vue métier, indépendamment des conditions d'exécution. Elle représente l'implémentation des interfaces fournies définies dans un type de composant. Dans l'exemple de l'agenda, l'implantation fonctionnelle regroupe la réalisation des fonctionnalités telles l'ajout ou le retrait de rendez-vous. L'implantation non fonc-

tionnelle regroupe quant à elle les besoins en terme de services techniques (transactions, sécurité, persistance, ...). Par exemple, on peut souhaiter que les fonctionnalités d'ajout et de retrait de rendez-vous de l'agenda soient sécurisées (contrôle d'accès) et exécutées dans un contexte transactionnel. L'implantation fonctionnelle est en règle générale programmée alors que l'implantation non fonctionnelle qui représente l'adaptation de ce code métier aux conditions d'exécution est quant à elle décrite et sa prise en charge automatisée.

Instance de composant - Une instance de composant est, au même titre qu'un objet, une entité existante et s'exécutant dans un système. Elle est caractérisée par une référence unique, un type de composant et une implantation particulière de ce type. De même que les objets, une instance de composant peut recevoir des invocations de méthodes. Une différence importante entre instance de composant et objet repose sur l'utilisation du concept de *fabrique* (ou Home) qui permet de gérer le cycle de vie des instances de composants. Dans la suite, à chaque fois que nous utilisons le terme « composant », nous faisons référence à l'instance et non à son type.

Un ensemble de modèles et plates-formes à composants sont décrits dans les sections suivantes. Les travaux autour du modèle de composants CORBA [89], celui de Fractal [86] ainsi que Sofa et Molène [109] ont été choisis pour leurs différences en terme d'architecture et de processus de mise en œuvre des adaptations dynamiques.

2.2.2 CCM

Spécifié par l'OMG (Object Management Group) en 2002, le modèle à composants CCM (CORBA Component Model) ajoute une couche au dessus de l'intergiciel CORBA (Common Object Request Broker and Architecture) [92] permettant de définir l'architecture d'une application distribuée sous forme de composition de composants. La description d'une composition, appelée *Assembly*, est réalisée de façon déclarative et permet de guider le déploiement des types de composants ainsi que la création, configuration, connexion et activation des instances de ces derniers.

Notons que le standard EJB [104] est un modèle de composants logiciels répartis, reposant sur une technologie de conteneurs similaire à celle du modèle CCM. Les EJBs ont ouvert la voie vers un réel monde de composants à travers un nombre important d'implémentations dont Jonas [87] et JBoss [57]. Cependant, le modèle des EJB ne représente qu'un sous-ensemble du modèle CCM (entre autre, l'enfouissement de la connectique au sein même des implantations est un point faible du modèle EJB) et ne permet pas d'adaptations dynamiques. Il n'est donc pas étudié dans cette thèse mais nous y faisons référence si nécessaire.

Les types de composants CCM sont décrits en utilisant une extension de l'IDL (Interface Definition Language) de CORBA nommée IDL3 qui permet de définir les types indépendamment de toute considération liée à leurs implantations. Un type de composant CCM est constitué d'un ensemble de *ports* (i.e. interfaces) dont il existe quatre variétés : les *facettes* et les *réceptacles* (interfaces synchrones fournies et requises) et les *puits* et les *sources* d'évènements (interfaces asynchrones fournies et requises). Chaque facette / puits regroupe une partie des fonctionnalités disponibles sur le type de composant et peut représenter un point de vue sur le composant si celui-ci est segmenté. La notion de point de vue permet à chaque utilisateur de manipuler un système avec une vision qui lui est propre, dans ce cas, deux fonctionnalités de même signature dans deux facettes distinctes peuvent correspondre à un code différent. Un composant ne peut pas être accédé directement mais uniquement par l'intermédiaire de ses facettes. Enfin, les interfaces ont une cardinalité (simple ou multiple) permettant d'indiquer le nombre de connexions [8] qui peuvent être réalisées vers un composant.

Par exemple, le type IDL du composant `Agenda` est défini par la figure 2.5. Les facettes `RDVConsultFacet` et `RDVModifFacet` correspondent à des interfaces permettant respectivement de consulter et modifier les rendez-vous associés à l'agenda courant. Ces facettes représentent respectivement le point de vue du possesseur de l'agenda et de son entourage qui y a accès. A titre

[8] Une connexion est établie soit entre une facette et un réceptacle soit entre une source et un puit.

d'exemple, l'interface `RDVConsultFacet` est détaillée dans la figure 2.5. Le réceptacle `AffichageFacet` indique que l'agenda utilise un composant d'affichage fournissant l'interface `AffichageFacet`. Enfin, le composant d'agenda émet des évènements de type `StockageEvt` pour communiquer avec un composant de base de données qui écoute ce type d'évènements.

```
component Agenda {
   provides RDVConsultFacet consult;
   provides RDVModifFacet modif;
   uses AffichageFacet afficheur;
   emits StockageEvt stock;
};
```

```
interface RDVConsultFacet {
   RdvList getRdvs();
   void getRdv(in Date d);
   void isFree (in Date d);
};
```

FIG. 2.5 – Exemple de définition du type de composant `Agenda` en CCM

De manière abstraite, les assemblages entre des composants de type `Agenda`, `Base de données` et `Afficheur` se présentent comme montré sur la figure 2.6.

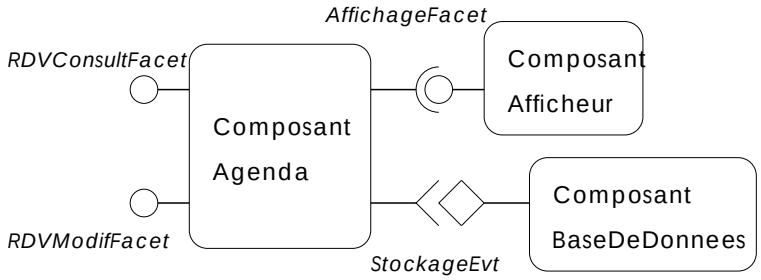

FIG. 2.6 – Assemblage d'un agenda avec une base de données et un afficheur en CCM

Les implantations de facettes et de puits d'événements représentent l'implantation fonctionnelle d'un composant. Cette partie de l'implantation est la seule à être programmée. La partie non fonctionnelle des implémentations de composants est décrite et générée à l'aide du langage de description CIDL (Component Implementation Language). Les *descripteurs de composants* (générés en grande partie à partir des types IDL et des définitions CIDL) contiennent des détails sur la structure logicielle (correspondances entre les ports définis dans le type du composant et un code d'implémentation de ce port, segmentation) sur leur catégorie (comme dans le modèle EJB [104], nous pouvons distinguer les entités sans état ou avec état non partagé ou avec état partagé) et sur la gestion des services techniques du composant. Ainsi, le code d'implantation des composants concerne uniquement la logique métier. Quant aux *descripteurs d'assemblages*, ils décrivent les architectures initiales des applications. Ils spécifient les composants constituant une application, définissent des règles de placement sur des sites d'exécution et donnent les connexions à établir entre instances.

Les *containers* sont des environnements d'exécution pour les composants qui ont la particularité de prendre implicitement en charge la partie non fonctionnelle de l'implantation des composants. Ces containers utilisent les descripteurs de composants pour fournir correctement les services techniques aux composants. Les composants sont créés à travers des fabriques appelées des *homes* et, une fois créés, leur cycle de vie est géré par leur container.

Dans l'exemple des agendas, l'assemblage d'un composant `agendaDeAM` de type `Agenda` avec un composant `dbms` de type `BaseDeDonnées` et un composant `emacs` de type `GUI` pourrait être décrit par le descripteur d'assemblage de la figure 2.7.

```
<componentassembly id="XYZ:0987654321">
  <description> Exemple de descripteur d'assemblage de composants </description>
  <componentfiles> ... </componentfiles>
  <partitioning> ... </partitioning>
  <connection>
    <connectevent>
      <consumesport>
         <consumesidentifier> StockageEvt </consumesidentifier>
         <componentinstanciationref idref="agendaDeAM">
      </consumesport>
      <publishesport>
         <publishesidentifier> StockageEvt </publishesidentifier>
         <componentinstanciationref idref="dbms">
      </publishesport>
    </connectevent>
    <connectinterface>
      <usesport>
         <usesidentifier> AffichageFacet </usesidentifier>
         <componentinstanciationref idref="agendaDeAM">
      </usesport>
      <providesport>
         <providesidentifier> AffichageFacet </providesidentifier>
         <componentinstanciationref idref="emacs">
      </providesport>
    </connectinterface>
  </connection>
</componentassembly>
```

FIG. 2.7 – Description des connexions de l'agenda avec l'afficheur et la base de données en CCM

Le modèle de composants CORBA impose que tous les ports d'un composant soient déclarés statiquement à la définition du type de composants. Ainsi, il est impossible d'ajouter ou retirer des ports dynamiquement durant l'exécution. Cependant, bien que les informations décrites dans les descripteurs d'assemblages soient statiques (définies à l'avance, avant le déploiement), il est possible de changer les connexions durant le cycle de vie d'un composant. Il est possible de supprimer une connexion entre deux composants, d'ajouter une connexion entre deux composants[9] (et donc de remplacer un composant par un autre en combinant les deux types d'adaptations élémentaires précédents) qui sont des types d'adaptations élémentaires appartenant à la famille **altération des assemblages**.

Le code de la figure 2.8 montre comment remplacer, à l'exécution, le composant `dbms` de type `BaseDeDonnées` par le composant `ntfs` de type `SystèmeDeFichier` dans la connexion `StockageEvt` qui lie la base de données au composant `agendaDeAM` dans une plate-forme respectant le modèle CCM telle que OpenCCM [72]. Notons que les opérations d'introspection (`get_stock_consumer()`) et les opérations de gestion des connexions (`disconnect_stock()` et `connect_stock()`) sont générées automatiquement pour chaque port à partir des interfaces IDL.

```
agendaDeAM.disconnect_stock()
agendaDeAM.connect_stock(ntfs.get_stock_consumer())
```

FIG. 2.8 – Modification des connexions de l'agenda en CCM

[9] A condition que ce type de connexion soit prévu statiquement sous forme d'un réceptacle / puits.

Par contre, il est impossible de créer de nouvelles connexions (qui n'auraient pas été prévues) ou d'en supprimer (i.e. : on ne peut pas changer le type d'un composant pour ajouter ou supprimer des interfaces requises). Dans l'exemple, il aurait été impossible de créer dynamiquement une connexion entre un composant de type `GUI` et un composant `Agenda` si ce type de connexion n'a pas été prévu (via le réceptacle `afficheur` de type `AffichageFacet` coté `Agenda` et une facette de type `AffichageFacet` coté `GUI`) à la définition du type de composant `Agenda`.

2.2.3 Fractal

Fractal [86] est un modèle à composants qui ne présuppose pas une sémantique ou une granularité particulière associée aux composants à l'inverse des modèles de composants industriels tels que EJB [104] ou CCM [89] qui modélisent uniquement des composants « métiers » dont les « containers » fournissent des services techniques. Fractal fournit une API permettant de créer, introspecter et gérer les composants, leurs interfaces et leurs liaisons.

Un type de composant (*ComponentType*) est constitué par un ensemble d'interfaces de deux catégories : les interfaces de contrôle et les interfaces fonctionnelles.

- Les interfaces de *contrôle*, liées à la gestion des besoins non fonctionnels du composant (gestion du cycle de vie, des liaisons entre composants, ...), sont les points d'accès aux contrôleurs d'un composant (les contrôleurs sont détaillés ci-après).
- Les interfaces *fonctionnelles* sont les points d'accès externes aux fonctionnalités d'un composant : tout comme les composants CORBA, un composant Fractal ne peut être utilisé qu'à travers des références sur ses interfaces fonctionnelles. Les interfaces fonctionnelles de type *serveur* (interfaces fournies) reçoivent des invocations et les interfaces fonctionnelles de type *client* (interfaces requises) émettent des invocations. Les interfaces fonctionnelles peuvent de plus être catégorisées comme étant *obligatoires* ou *optionnelles* et ont une cardinalité simple (*singleton*) ou multiple (*collection*). Par exemple, le type de composant `AgendaType` comporte l'interface fonctionnelle serveur obligatoire `AgendaDeBaseItf` (qui définit les méthodes `addRdv`, `removeRdv` et `getRdvs`) comme le montre la figure 2.9.

```
ComponentIdentity bootstrap = Fractal.getBootstrapComponent();
TypeFactory typeFactory = (TypeFactory) bootstrap.getFcInterface("type-factory");
ComponentType agendaType = typeFactory.createFcType(new InterfaceType[]{
        typeFactory.createFcItfType("server", "AgendaDeBase", TypeFactory.SERVER,
                                    TypeFactory.MANDATORY, TypeFactory.SINGLE)});

Interface AgendaDeBase {
   void addRdv(Rdv);
   void removeRdv(Rdv);
   RdvList getRdvs();
}
```

FIG. 2.9 – Définition du type de composant Agenda en Fractal

Il existe deux catégories de composants : les composants *primitifs* et les composant *composites*. Et l'implémentation d'un composant est divisée en deux parties : le *contenu* et la *membrane*.

- Le contenu d'un composant primitif comporte une brique d'implémentation (du code fonctionnel sous la forme d'un objet) alors que le contenu d'un composant composite comporte d'autres composants (primitifs ou composites). Notons que le modèle est complètement récursif et autorise l'imbrication des composants à un niveau arbitraire. Cette spécificité fait de Fractal un modèle hiérarchique contrairement aux EJB et CCM qui sont des modèles « plats ».

– La membrane d'un composant (primitif ou composite) comporte un ensemble d'*intercepteurs* et de *contrôleurs*. Les intercepteurs interceptent certaines méthodes et les redirigent vers les contrôleurs auxquels ils sont associés. Les contrôleurs sont les points d'entrée pour l'acquisition de services techniques pour un composant donné. Un composant composite permet donc de gérer la « composition » de différentes briques logicielles au moyens de ses propres contrôleurs ou en délégant une partie du travail aux sous-composants. Lorsque les contrôleurs d'un composite agissent sur son contenu (i.e. : ses sous-composants), le composite joue un rôle similaire à celui d'un container aux sens des EJBs ou du CCM.

Un composant d'agenda peut s'écrire en Fractal comme un composite dont le contenu est constitué de deux composants primitifs définis à partir des classes Java `AgendaDeBaseImpl` et `GraphicalPrinterImpl` par exemple. La figure 2.11 décrit l'architecture d'un tel type de composant. La définition de cette architecture peut s'effectuer grâce à l'ADL (Architecture Description Language) [76] Fractal (l'équivalent des descripteurs d'assemblages du modèle CCM) ou bien être programmée. La figure 2.10 illustre les instructions nécessaires pour obtenir le composite.

```
GenericFactory gf = (GenericFactory) bootstrap.getFcInterface("generic-factory");

// Création des fabriques d'afficheurs, d'agendas primitifs et d'agendas composites
ComponentIdentity agendaInterneTemplate =
    gf.newFcInstance(agendaInterneType,"template", new Object[]{"primitive","AgendaDeBaseImpl"});

ComponentIdentity printerTemplate =
    gf.newFcInstance(printerType,"template", new Object[]{"primitive","GraphicalPrinterImpl"});

ComponentIdentity agendaTemplate =
    gf.newFcInstance(agendaType,"compositeTemplate", new Object[]{"composite",null});

ContentController agendaTemplateCC =
    (ContentController) agendaTemplate.getFcInterface("content-controller");

// Composite agenda constitué de primitifs agenda et afficheur graphique
agendaTemplateCC.addFcSubComponent(agendaInterneTemplate);
agendaTemplateCC.addFcSubComponent(printerTemplate);

// Connexion entre composite agenda et primitif agenda
((BindingController) agendaTemplate.getFcInterface("binding-controller"))
.bindFc("server", agendaInterneTemplate.getFcInterface("server"));

// Connexion entre primitif agenda et primitif afficheur graphique
((BindingController) agendaInterneTemplate.getFcInterface("binding-controller"))
.bindFc("refresh", printerTemplate.getFcInterface("server"));
```

FIG. 2.10 – Création d'un agenda en Fratal

Les composants sont créés à partir de fabriques[10]. Un composant peut être activé ou désactivé soit par son propre contrôleur si la membrane comporte un contrôleur implémentant l'interface de contrôle `LifeCycleController` soit par un contrôleur de niveau supérieur (contrôleur d'un composant plus englobant). Dans la suite supposons que `agendaTemplate` nous ait permis de créer l'instance `agendaDeAM`.

[10]La fabrique fournit une méthode de création (`instantiateFc()`) qui retourne une nouvelle instance à chaque invocation.

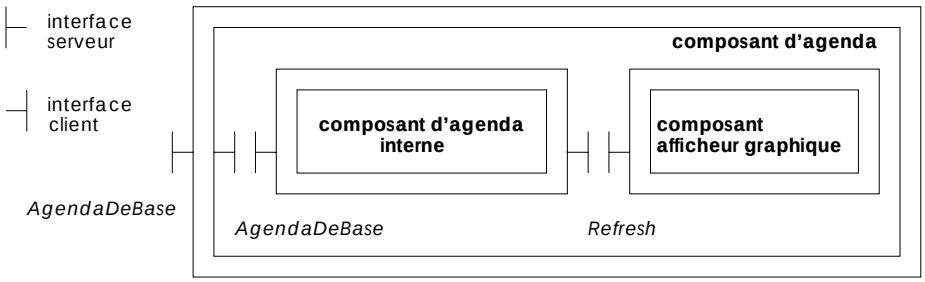

FIG. 2.11 – Architecture Fractal d'un agenda

Julia [21], l'implémentation Java de référence de fractal, permet l'**altération des assemblages** de manière explicite et dynamique. Cette famille d'adaptations élémentaires est possible du fait que les composants composites peuvent modifier leur contenu à l'exécution. Il est possible de connecter dynamiquement deux composants, de les déconnecter (et donc aussi de remplacer un composant par un autre en combinant les deux types d'adaptations élémentaires précédents). En fait, on peut soit remplacer le contenu /implantation d'un composant soit le composant tout entier. Par contre, tout comme dans le modèle des composants CORBA, il est impossible de créer de nouvelles connexions ou d'en supprimer (i.e. : on ne peut pas changer le type d'un composant). La figure 2.12 montre le cas où seule l'implantation est changée. Dans cet exemple, on change la manière dont l'agenda est affiché.

```
// Arrêt de l'instance agendaDeAM
((LifeCycleController) agendaDeAM.getFcInterface("lifecycle-controller")).stopFc();

// Déconnexion de l'agenda interne de agendaDeAM de son afficheur
((BindingController)cache.getFcInterface("binding-controller")).unbindFc("refresh");

// Retrait de l'afficheur du contenu de agendaDeAM
ContentController agendaDeAMCC = (ContentController) agendaDeAM.getFcInterface("content-controller");
agendaDeAMCC.removeFcSubComponent(printer);

// Nouvelle fabrique d'afficheurs
ComponentIdentity newPrinterTemplate =
    gf.newFcInstance(printerType,"template", new Object[]{"primitive","TextualPrinterImpl"});

// Une instance d'afficheur textuel
ComponentIdentity newPrinter =
    ((Template)newPrinterTemplate.getFcInterface(Template.TEMPLATE)).instantiateFc();

// Ajout de cette instance dans le contenu de agendaDeAM
agendaDeAMCC.addFcSubComponent(newPrinter);

// Connexion entre l'agenda interne de agendaDeAM et le nouvel afficheur
((BindingController)agendaInterne.getFcInterface("binding-controller"))
.bindFc("refresh", newPrinter.getFcInterface("server"));

// Redémarrage de l'instance agendaDeAM
((LifeCycleController) agendaDeAM.getFcInterface("lifecycle-controller")).startFc();
```

FIG. 2.12 – Reconfiguration d'un agenda en Fractal

Julia permet aussi la **composition comportementale** du fait que les contrôleurs peuvent changer le comportement associé aux fonctionnalités d'un composant, cependant cette famille de types d'adaptations élémentaires est mise en œuvre uniquement statiquement dans Julia c'est à dire que l'on ne peut pas changer, au cours de son cycle de vie, les contrôleurs associés à un composant. Nous expliquons en 2.3 comment allier les possibilités en terme d'adaptations des approches à composants et des approches orientées aspects.

2.2.4 D'autres modèles en bref

Dans les deux sections précédentes, nous avons vu des modèles à composants très différents en terme d'architecture : CCM, un modèle à composants plat, et Fractal, un modèle à composants hiérarchique. Cette section présente les spécificités d'autres modèles méritant d'être soulignées.

SOFA

SOFA / DCUP (SOFtware Appliances / Dynamic Component UPdating) [101] est un modèle à composants développé à l'université Charles (Prague). DCUP est un modèle hiérarchique tout comme Fractal [86], permettant l'adaptation sous forme de mise à jour dynamique des composants. SOFA vise la transparence de l'adaptation dynamique, la prise en compte de l'état d'exécution pendant l'adaptation, et le support de téléchargement automatique et dynamique des composants. Une application DCUP est une hiérarchie de composants interconnectés, un composant lui-même pouvant être construit par assemblage de sous-composants.

La force de SOFA réside dans la définition de protocoles comportementaux [3]. Les protocoles sont des contrats de synchronisation [15] définis à plusieurs niveau pour décrire comment les composants doivent être utilisés : le comportement d'un composant SOFA correspond à l'ensemble des enchaînements d'invocation de méthodes autorisées.

- Les protocoles d'interface (*interface protocol*) décrivent comment utiliser une interface.
- Les protocoles de « composition » (*Frame protocol*) décrivent comment assembler les différentes interfaces d'un composant et celles de ses composants internes.

Les protocoles d'interface correspondent en général à l'exécution de n'importe quelle méthode dans n'importe quel ordre sauf si, par exemple, l'interface comporte des méthodes d'initialisation et de terminaison[11]. Les protocoles de composition permettent de construire un composant à partir d'autres composants en s'appuyant sur les protocoles de ces derniers. L'utilisation de protocoles comportementaux pour définir la composition des composants est le principal point qui différencie ce modèle de Fractal.

Reprenons l'exemple de l'agenda et considérons le cas où il est connecté à un composant de base de données. Les protocoles des interfaces `AgendaDeBase` et `Persistance` indiquent que n'importe quelle méthode peut être exécutée dans n'importe quel ordre (cf figure 2.13). Le protocole défini entre les interfaces du composant d'agenda (`AgendaFrame`) spécifie comment les appels entre les fonctionnalités fournies et requises du composant s'entrelacent. Le symbole "?" devant un nom de méthode désigne la réception et "!" désigne l'appel. Ainsi, le protocole de `AgendaFrame` indique que la réception de `addRdv` ou `removeRdv` sur l'agenda est suivie d'un appel à `store` sur le composant de base de données (l'opérateur ";" indique le séquencement) et que la réception de `getRdvs` sur l'agenda dépend de l'appel à `load` sur le composant de base de données (l'utilisation des accolades indique la dépendance). L'opérateur "|" indique que les deux scénarios peuvent se produire en parallèle et le "*" que ces deux scénarios peuvent être répétés indéfiniment.

[11] Une interface de manipulation d'un système de fichier impose que l'ouverture d'un fichier précède toutes les autres actions, que la fermeture soit la dernière et entre les deux une suite quelconque de lecture et d'écriture : (open ; (read+write)* ; close).

```
interface AgendaDeBase {
   void addRdv(Rdv);
   void removeRdv(Rdv);
   RdvList getRdvs();
  protocol:
     (addRdv + removeRdv + getRdvs)*
};
```

```
interface Persistance {
   void store(String, Component);
   Component load(String);

  protocol:
    (store + load)*
};
```

```
frame AgendaFrame {
  provide:
       AgendaDeBase a;
  require:
       Persistance db;
  protocol:
     ((?a.addRdv + ?a.removeRdv); !db.store)* |  (?a.getRdvs{!db.load} )*
};
```

FIG. 2.13 – Exemple de protocoles comportementaux Sofa pour l'agenda

Une des particularités de Sofa est qu'un composant est constitué d'une partie permanente (interfaces et protocoles) et d'une partie remplaçable (contenu / implantation). L'adaptation dynamique d'un composant revient donc à changer sa partie remplaçable par une nouvelle version. Ainsi, les opérations d'adaptation explicites sont réduites au remplacement de l'implantation des composants qui fait partie de la famille **altération des assemblages**. L'interface du composant, comprise dans la partie permanente, ne peut alors pas être changée. Toutefois, si on souhaite ajouter un composant, ou changer une connexion à un certain niveau, le seul moyen possible est de redéployer tout le composant englobant le niveau en question. Ceci est très coûteux, et peut impliquer l'adaptation de toute l'application, si le changement doit intervenir au plus haut niveau de la hiérarchie des composants.

OSGi

OSGi (Open Services Gateway initiative) [93] est un modèle à composants dont la particularité est d'être orienté services c'est à dire à dire qu'il permet de mettre en œuvre des applications suivant la SOA. Une architecture orientée services (SOA pour Service Oriented Architecture) [84] est fondée sur la définition de services coopérants. Un service correspond à une fonctionnalité dont le comportement est défini par un contrat d'utilisation et est fourni par une entité nommée *fournisseur de service*. Les fournisseurs de service sont découverts à l'exécution à l'aide d'un intermédiaire. Une application construite à partir de services utilise un ensemble de services et les fournisseurs de ces services ne sont pas câblés dans l'application : un service n'appelle jamais directement un autre service et les fournisseurs peuvent changer à travers les différentes exécutions de celle-ci. De nouveaux services peuvent être découverts (via leur contrat) et invoqués à l'exécution. Les travaux autour des services web [64] entrent également dans le cadre de la SOA.

Dans la spécification OSGi, les services sont livrés et déployés dans des unités appelées *bundles*. Un bundle correspond à un demandeur ou fournisseur de services c'est à dire, plus généralement, à un composant. La spécification définit des mécanismes d'administration permettant de réaliser l'installation, activation, désactivation, mise à jour et désinstallation des composants de façon continue.

Le type des composants n'est pas explicite en OSGi. En effet, il est possible de décrire l'ensemble des services fournis et requis par un composant. Cependant ces informations sont facultatives et ne sont mentionnées qu'à titre informatif. Un bundle est concrètement constitué d'un ensemble de classes et d'interfaces Java, d'un manifeste et d'un ensemble de ressources (pages HTML, images). Par défaut, un type de composant n'est instancié qu'une seule fois. Cette instance est partagée par tous les autres composants qui en ont besoin. Pour changer ce fonctionnement par défaut, et pour pouvoir contrôler le nombre d'instances à créer, le composant doit définir une fabrique en implémentant l'interface `ServiceFactory`.

L'utilisation d'événements permet de suivre l'évolution de l'application et de réagir aux changements qui se produisent. Lorsque un composant est actif, il peut publier ou découvrir des services et se lier avec d'autres composants à travers un registre de services. Un mécanisme de composition de services, appelé *Wire Admin Service*, permet de lier des producteurs (générant de l'information) à des consommateurs (recevant de l'information) à l'aide de connecteurs appelés *wires*. Un connecteur décrit une association entre un producteur et un consommateur et plusieurs connecteurs peuvent lier un producteur et un consommateur. Lorsque les deux entités deviennent disponibles, un administrateur de connecteurs (*wire admin*) est chargé de les lier à travers des connecteurs. La liaison entre producteurs et consommateurs est donc définie de façon statique, cependant les connexions peuvent apparaître ou disparaître pendant l'exécution selon la présence des producteurs et des consommateurs. Deux types d'adaptations élémentaires sont donc pris en charge explicitement et dynamiquement : la connexion et la déconnexion de composants qui font partie de la famille **altération des assemblages**.

Molène

Molène (MObiLE Networking Environement) [109] est un framework essentiellement destiné à l'auto-adaptation dynamique des composants en environnements mobiles. La particularité du framework par rapport aux autres approches étudiées est de fournir la possibilité de s'auto-observer et d'observer l'environnement d'exécution, de détecter des changements significatifs et de se reconfigurer en conséquence.

L'approche se base sur un gestionnaire de reconfiguration qui se charge ensuite d'opérer les reconfiguration sur les composants de l'application lorsque des changements lui sont notifiés. Les composants Molène réifient leurs besoins en terme d'adaptation par un automate dont les états représentent des conditions d'exécution et dont les transitions représentent une variation significative de l'environnement et permettent de déterminer l'adaptation à effectuer lorsque ces variations sont observées.

Un seul type d'adaptation élémentaire est pris en charge explicitement et dynamiquement par le framework : le remplacement de l'implantation des composants qui fait partie de la famille **altération des assemblages**.

2.3 Approches hybrides

Nous pouvons identifier deux points communs aux approches par composants et aux approches orientées aspects. D'une part, la modification comportementale, offerte par les approches orientées aspects, induit implicitement une modification du graphe d'interactions entre les différentes entités logicielles : une altération des assemblages si on se rapproche du vocabulaire des composants. Cependant, ces assemblages ne sont pas manipulables explicitement dans les approches orientées aspects.

D'autre part, les notions de containers (CCM) et de contrôleurs (Fractal) traduisent la volonté dans les approches à composants de sortir de l'implantation des composants, le code non fonctionnel lié aux services (sécurité, transactions, persistance, ...), c'est à dire de mettre en œuvre la séparation des préoccupations dans une certaine mesure. Néanmoins, cette mise en œuvre reste

très limitée : ces services techniques sont câblés en dur dans le code source du container et sont en nombre prédéfini que ce soit dans le cas de plates-formes respectant la spécification CCM telles que OpenCCM comme pour les plates-formes respectant la spécification EJB telles que Jonas. Quant au principe des contrôleurs, il permet l'acquisition de n'importe quel service puisqu'il est possible d'écrire ses propres contrôleurs. Cependant, comme nous l'avons dit dans la section 2.2.3, l'ensemble des contrôleurs associés à un composant est spécifié statiquement et ne peut être modifié durant l'exécution.

Une nouvelle tendance consiste alors à unifier les deux approches pour permettre d'élargir le spectre de types d'adaptations élémentaires qu'elles prennent en charge. Quelques travaux commencent à émerger dans ce domaine. Nous présentons dans cette section le modèle FAC [100] qui nous semble être le projet le plus abouti.

FAC

FAC (Fractal Aspect Component) [100] est une extension du modèle de composants Fractal pour la programmation par aspects. Ce modèle permet de tisser des aspects à une application programmée à l'aide de composants Fractal. Pour cela, l'extension enrichit un assemblage classique de composants métiers Fractal avec un assemblage implémentant des préoccupations transverses.

Un nouveau type de composant appelé composant d'aspect[12] est défini pour décrire le code de traitement (advice) d'un aspect (composant primitif). Les composants d'aspect sont hiérarchiques comme les composants Fractal classiques. En particulier, un composant d'aspect composite peut utiliser (i.e. : avoir dans son contenu) des composants Fractal classiques pour réaliser le code de traitement de l'aspect qu'il décrit. Ce qui caractérise les composants d'aspect est le fait qu'ils doivent implémenter la méthode `invoke` de l'interface serveur `AspectComponent` qui définit réalisant le code de l'advice.

Tisser un aspect à une application à base de composants Fractal avec FAC revient à connecter un composant d'aspect à un composant Fractal classique pour en modifier le comportement. Pour cela, un nouveau type de liaison, nommé *liaison transverse*, est introduit pour « lier » ces deux types de composants (cette liaison peut être multiple lorsque le même aspect impacte plusieurs composants). Le tissage devient alors une activité d'assemblage classique sauf que dans ce cas précis, on ne lie pas une interface fonctionnelle à une autre interface fonctionnelle mais à une interface de contrôle. Il existe deux façons de réaliser la liaison entre un composant Fractal et un composant d'aspect : soit directement entre les deux composants en utilisant la méthode `bindAC` dans ce cas toutes les méthodes du composant seront interceptées, soit de manière déclarative en utilisant la méthode `wave` sur le composant racine (de plus haut niveau) de l'application et en spécifiant la coupe. Le langage de coupe correspond à trois expressions régulières sur les noms de composants, d'interfaces et de signatures de méthodes. Toutes les méthodes qui vérifient ces trois expressions sont liées au composant d'aspect. Enfin, ce modèle offre des possibilités en terme d'introspection de coupes pour permettre de connaître l'ensemble des aspects s'appliquant sur un composant donné ou l'ensemble des composants affectés par un composant d'aspect.

La plate-forme Julius [99] est une implémentation du modèle FAC au dessus de Julia et met en œuvre l'API AOP Alliance [7] tout comme JAC. Julius définit un contrôleur d'interception Fractal / Julia pour gérer le ou les aspects tissés sur un composant. Ce contrôleur a la charge d'intercepter les appels entrant et sortant d'un composant et de les rediriger vers des composants d'aspects. Pour connecter un composant Julius à un composant d'aspect, ce dernier doit donc comporter le contrôleur d'interception.

Julius permet de réaliser explicitement et dynamiquement deux types d'adaptations élémentaires : la modification du comportement d'une fonctionnalité existante d'un objet appartenant à la famille **composition comportementale** et la modification du contenu d'un composant appartenant à la famille **altération des assemblages**. Ces adaptations élémentaires sont mises en œuvre

[12] Le terme provient des travaux sur JAC.

de la même manière : par la manipulation des liaisons entre composants. Revenons à l'exemple de l'agenda. Nos agendas sont des composants Fractal (voir l'exemple dans la section de ce chapitre qui traite de Fractal). La figure 2.14 décrit un assemblage avec liaison transverse. Le composant de trace est un composant d'aspect primitif permettant de tracer l'exécution de certaines méthodes d'un composant auquel il est lié par une liaison transverse en modifiant le comportement des méthodes en question. En l'occurrence, la liaison entre les composants a été réalisée en utilisant la méthode `bindAC`, le composant de trace va donc tracer toutes les méthodes de l'agenda. La figure 2.15 montre le code de la méthode `invoke` du composant de trace.

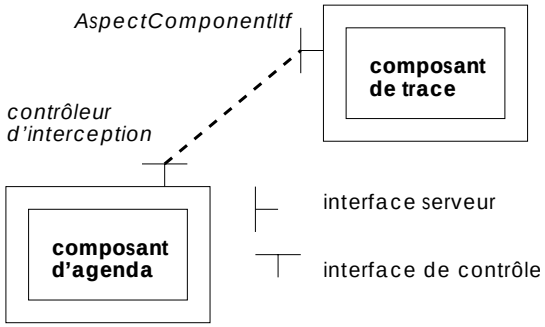

FIG. 2.14 – Exemple d'assemblage avec liaison transverse dans FAC

```
import org.aopalliance.intercept.MethodInvocation;
...
public Object invoke(MethodInvocation mi) throws Throwable {
  Object ret = null;
  System.out.println("<< calling "+mi.getMethod().getName()+
                     " with "+mi.getArguments()+">>");

  // passe la main au prochain composant d'aspect
  // ou exécution de la méthode initiale
  ret = proceed(mi);

  System.out.println("<< "+mi.getMethod().getName()+" returned "+ret+">>");
  return ret;
}
```

FIG. 2.15 – Exemple de code pour l'advice de trace dans FAC

2.4 Synthèse

Le tableau 2.1 récapitule quels sont les types d'adaptations élémentaires qui sont mis en œuvre dans chaque plate-forme que nous avons détaillée auparavant. Si on regarde les différentes plates-formes permettant d'effectuer des adaptations dynamiques, on se rend compte qu'elles ne prennent pas en charge les mêmes types d'adaptations élémentaires. Dans les approches par composants, on va pouvoir modifier les assemblages de composants de manière explicite via une API dédiée. Alors que dans les approches orientées aspects, ce qui est mis en avant est de pouvoir changer le comportement des entités logicielles ou encore leurs interfaces.

Cependant, la modification comportementale via les aspects, induit implicitement une modification du graphe d'interactions entre les différentes entités logicielles : une altération des assemblages si on se rapproche du vocabulaire des composants. A l'inverse, l'altération des assemblages de composants induit implicitement une modification du comportement des composants. Ainsi, en plus des types d'adaptations mis en œuvre explicitement, il existe des types d'adaptations cachés. Enfin, on se rend également compte que les adaptations élémentaires ne sont pas mises en œuvre de la même manière, elles peuvent être spécifiées ou programmées.

	Familles d'adaptation		
	Evolution des interfaces	**Composition comportementale**	**Altération des assemblages**
JAC	Ajout de fonctionnalités	Chaînage de wrappeurs	
CF / Compose*	Ajout et retrait de fonctionnalités	Superimposition de filtres	
Noah		Fusion de règles de réécriture	
CCM / OpenCCM			Connexion et déconnexion de composants
Fractal / Julia			Connexion et déconnexion de composants
Sofa / DCUP			Remplacement de composants (implantation)
OSGi			Connexion et déconnexion de composants
Molène			Remplacement de composants (implantation)
FAC / Julius		Connexions transverses	Connexion et déconnexion de composants

TAB. 2.1 – Types d'adaptation élémentaires prises en charge explicitement et dynamiquement par différentes plates-formes

Dans le chapitre suivant nous étudions quels types de problèmes de sûreté émergent en raison des adaptations dynamiques des applications et comment les plates-formes étudiées répondent à ces problèmes.

« Une erreur ne devient une faute que lorsqu'on ne veut pas en démordre »

Ernst Jünger

« D'idées vraies en idées vraies et de clartés en clartés, le raisonnement peut n'arriver qu'à l'erreur »

Rivarol

3

Sûreté de fonctionnement

La *sûreté de fonctionnement* des systèmes informatiques est définie par Laprie [56] comme étant la propriété permettant aux utilisateurs d'un système de placer une confiance justifiée dans le service qu'il leur délivre. Ainsi, un système n'est plus « sûr » à partir du moment où le service qu'il fournit diverge du service attendu par l'utilisateur. Cette divergence entre ce qui est fourni et ce qui est attendu est la conséquence d'une *faute* commise par une partie du système qui génère donc des *erreurs* ou *incohérences*.

Si un humain est capable de concevoir et d'écrire un algorithme, il lui est en revanche plus difficile de l'exécuter mentalement. A l'inverse, un ordinateur exécute fidèlement les instructions décrites par cet algorithme, y compris lorsque celles-ci conduisent à une situation que l'utilisateur n'a pas souhaitée, mais dont il n'a pas su détecter l'éventualité. Cette difficulté à comprendre ce que fait un programme dès que celui-ci devient de taille non négligeable explique pourquoi toute application informatique contient un certain nombre de « bugs », c'est à dire d'erreurs programmées de manière involontaire qui conduisent à briser la sûreté de fonctionnement d'une application. Un certain nombre de techniques (typage, model-checking, ...) ont vu le jour pour pallier ce problème et fournir des outils pour bien concevoir et implémenter les systèmes informatiques.

Avec l'apparition des plates-formes permettant les adaptations dynamiques, la sûreté de fonctionnement des applications est de nouveau remise en cause quand bien même ces dernières ont été validées par les techniques sus citées. Trois facteurs peuvent rendre l'application adaptée non sûre. Premièrement, l'adaptation peut introduire un élément qui invalide la preuve sur la sûreté de fonctionnement de l'application initiale (chaque composant est sûr mais l'assemblage créé par l'adaptation ne l'est pas car une connexion est mal typée par exemple). Deuxièmement, l'adaptation peut se produire à un moment inadéquat dans l'exécution de l'application (c'est le cas si elle vient interrompre une transaction par exemple). Troisièmement, l'adaptation peut ne pas être mise en œuvre correctement (par exemple, en cas d'une panne du processus adaptant l'application, certaines modifications sont effectuées mais pas d'autres, laissant l'application dans un état inconsistant).

Dans la section 3.1, nous montrons quels types de problèmes de sûreté émergent en raison des adaptations dynamiques des applications et comment les plates-formes présentées dans le chapitre 2 répondent à ces problèmes. Dans un second temps, nous étudions dans quelles mesures des techniques destinées à fiabiliser la construction des applications peuvent aider à fiabiliser le processus d'adaptation dynamique des applications dans la section 3.2.

3.1 Sûreté dans les systèmes adaptatifs dynamiques

Plusieurs acteurs interviennent dans le processus de construction des applications (concepteur / architecte, développeur, intégrateur, déployeur, administrateur). L'apparition des systèmes adaptatifs dynamiques a introduit un rôle supplémentaire : l'adaptateur qui fait évoluer l'application durant son exécution. Notons que l'adaptateur n'est pas forcément une personne physique. Dans le cas d'applications sensibles au contexte (auto-adaptatives), c'est l'application elle même qui initie une évolution. Dans la suite, nous faisons référence à ces acteurs pour expliciter à qui incombe la responsabilité des vérifications en terme de gestion de la sûreté en fonction des systèmes adaptatifs.

3.1.1 Problèmes de sûreté liés aux adaptations dynamiques

Dans cette section, nous classifions différents problèmes de sûreté qui peuvent survenir lorsqu'on adapte une application à l'exécution pour répertorier les points forts et points faibles en terme de sûreté des différents systèmes adaptatifs dynamiques étudiés en 2.

Problème du typage dynamique

Définition du problème. Il existe deux sortes de typage. Lorsque le typage est statique, les types sont connus statiquement et ne peuvent pas être modifiés à l'exécution, les appels de méthodes sont résolus statiquement. Lorsque le typage est dynamique, aucune vérification de type n'est faite statiquement, n'importe quelle méthode peut être appelée en cours d'exécution y compris des méthodes auxquelles l'objet récepteur ne sait pas répondre. Par rapport aux adaptations dynamiques, nous pouvons distinguer deux sous-problèmes :

1. Comment prendre en compte l'ajout d'une fonctionnalité en cours d'exécution ?
   - Dans JAC, l'ajout de fonctionnalité n'est pas géré par une extension effective du type. Chaque message est intercepté à la réception sur l'objet destinataire. S'il ne s'agit pas d'une fonctionnalité offerte par l'objet (i.e. : elle n'appartient pas à son type), on va chercher à savoir si cette fonctionnalité a été ajoutée à l'objet par adaptation. Si ce n'est pas le cas, une erreur de type « message not understood » est déclenchée.
   - Le même mécanisme permet à Compose* d'ajouter des fonctionnalités : l'interception des messages permet à un objet d'accepter des messages dont l'implémentation peut en réalité être définie ailleurs, à l'extérieur de l'objet en question. Cette solution ne prenant pas en compte l'ajout de fonctionnalités comme une évolution de type, elle ne peut donc pas s'appuyer sur la vérification des types et détermine seulement si la fonctionnalité ajoutée est bien présente lorsque celle-ci est appelée.
2. Comment autoriser des appels de méthodes non connus statiquement ?
   - Le fait que le type des objets superimposés de Compose* n'est pas modifié lorsque les filtres permettent d'accepter de nouveaux messages pose un problème : comment compiler des classes qui référencent certaines méthodes introduites par un module de filtres. Le problème est résolu en construisant une classe racine de toutes les classes superimposées de l'application contenant l'union des messages interceptés par l'ensemble des modules définis et connus à la compilation de l'application (avec une implémentation par défaut la même pour toutes ces méthodes et qui consiste à lever une exception runtime[1]). Cette solution n'est pas tout à fait satisfaisante dans la mesure où elle restreint à la superimposition dynamique des modules uniquement connus statiquement.
   - Ce problème ne se pose pas pour JAC car chaque appel à une fonctionnalité ajoutée doit être effectué de manière explicite par invocation dynamique.

[1] Ceci est censé simuler le mécanisme du « message not understood » des langages non typés.

– Dans Noah, au moment du tissage d'un schéma, aucune vérification de type n'est faite. C'est à dire qu'on ne détecte pas si les fonctionnalités utilisées par les règles d'un schéma d'interactions sont fournies ou non par les entités sur lesquelles on tisse le schéma. Une erreur de type « message not understood » est seulement déclenchée au moment où on tente d'appeler ces fonctionnalités.

Problème de consistance des assemblages

Définition du problème. Comme les pièces d'un puzzle, les composants peuvent être assemblés de différentes manières mais toutes les possibilités en terme de composition ne sont pas valides. La « forme » d'une pièce de puzzle doit coïncider avec celle des pièces adjacentes et le puzzle n'est terminé que s'il ne manque aucune pièce. De la même façon, un assemblage de composants n'est pas consistant si au moins une connexion met en jeu deux composants dont les « formes » ne sont pas compatibles ou s'il manque un composant au bout d'une connexion.

Dans le modèle CCM, deux interfaces particulières sont utilisées pour assurer une bonne connexion entre les composants. L'interface d'introspection fournit des informations sur le type de composant, sur ses interfaces fournies et requises, ainsi que sur l'état de ses connexions à d'autres composants. L'interface de gestion de ports est utilisée pour établir ou détruire des connexions entre composants. Pour que les composants CCM d'une application puissent communiquer, leurs ports doivent être compatibles : une connexion est établie entre un port d'entrée et un ou plusieurs ports de sortie qui représentent des interfaces identiques et qui sont de la même nature (synchrone ou asynchrone). Si une demande de connexion ne satisfait pas les règles précédentes, elle est immédiatement refusée. De même, il existe des règles de typage sur les connexions en Fractal qui permettent de s'assurer que les ports sont compatibles.

Par contre, les assemblages partiellement connectés ne sont pas détectés en CCM. Par exemple, considérons un type de composant T comportant un port requis pr de type PR et un type de composant T' comportant un port fourni de type PR. Si un composant C de type T est connecté via PR à un composant C' de type T' alors lorsqu'on supprime la connexion entre C et C', aucune erreur n'est détectée à la déconnexion, c'est seulement au prochain appel à une fonctionnalité utilisant le port requis pr que le problème est soulevé, ce qui est bien trop tard.

Lorsqu'un composant OSGi est activé mais que les services dont il a besoin ne sont pas présents, le composant doit être mis en attente jusqu'à ce que ces services soient disponibles. Le même principe est utilisé lorsque des services utilisés par un composant d'une application bien assemblée disparaissent. Cette solution a comme désavantage le fait que le composant peut attendre indéfiniment l'apparition d'un autre service qu'il puisse utiliser à la place. Pour détecter les assemblages partiellement connectés, Fractal impose que toutes les interfaces clientes obligatoires d'un composant soient connectées à des interfaces compatibles avant que le composant puisse être (re)démarré.

Enfin, dans les modèles CCM, Fractal, FAC, OSGi et le framework Molène, remplacer une implantation ou un composant ne garantit pas que le comportement décrit par la nouvelle implantation ou le nouveau composant soit conforme au comportement décrit par l'implantation ou composant remplacé(e) car le comportement des composants n'est pas décrit. Grâce à la notion de protocoles, lorsqu'on remplace un composant SOFA, on peut vérifier que le nouveau composant peut être utilisé de la même manière que le composant remplacé.

Notons que d'autres travaux académiques se sont penchés sur le problème de consistance. Les travaux de Najm et al. [26] proposent des protocoles similaires à ceux de SOFA pour permettre le remplacement de composants avec une garantie de comportement et s'assurer que les assemblages ne sont pas partiellement connectés. Une autre implémentation de Fractal, BeFract / Brenda [11], vise à définir des composants dont l'activité est décrite par un automate pour permettre également le remplacement de composants avec une garantie de comportement.

Problème de cohérence de composition comportementale des adaptations

Définition du problème. Plusieurs adaptations visant à modifier le comportement d'une application peuvent intervenir en un même point (i. e. : même coupe). Il est donc nécessaire de pouvoir composer un ensemble d'adaptations. Différents types de conflits de composition ont été exposés dans la littérature [19, 97, 49, 73]. Par exemple, les conflits de type **exclusion mutuelle** interviennent lorsque la prise en compte d'une adaptation est en contradiction avec la prise en compte d'une autre adaptation. Les conflits de type **exécution conditionnelle** se produisent lorsque la prise en compte d'une adaptation dépend de la prise en compte d'une autre adaptation. Ou encore, les conflits de type **anomalies de composition** sont introduits lorsque la prise en compte d'une adaptation modifie un point de la structure de l'application utilisée par une autre adaptation déjà prise en compte.

Pour un grand nombre d'approches, composer des adaptations comportementales consiste donc à les ordonner pour réaliser le comportement « souhaité » :

- Lorsque plusieurs composants d'aspects FAC/Julius sont tissés, soit directement, soit par une expression de coupe sur le même composant, leur ordre d'exécution est celui de leur tissage c'est à dire que les composants d'aspect tissés en premier sont exécutés en premier.
- Même si elle est statique, la composition des contrôleurs dans Fractal/Julia suit également un ordre de composition qui correspond à l'ordre de déclaration des intercepteurs dans le fichier de configuration.
- La composition des filtres de composition est gérée implicitement par le langage de support en séquençant l'activation des filtres. L'ordre des filtres est important puisqu'un filtre peut masquer les filtres qui le suivent et ne pas leur transmettre certains messages. L'ordonnancement des filtres est réalisé manuellement par le développeur.

Or, tous les ordres de composition ne sont généralement pas équivalents et un certain nombre de conflits de composition peuvent survenir en fonction de l'ordre choisi. D'autre part, imposer un ordonnancement particulier implique de trouver un ordre total et même si celui-ci existe, ce n'est pas forcément la forme de composition recherchée. En effet, l'ordonnancent est une forme de composition de gros grain qui ne permet pas d'entrelacer intrinsèquement les adaptations comportementales.

JAC offre une approche plus complète au problème de la composition puisqu'il est non seulement possible de définir un ordonnancement mais également des politiques de composition qui dispensent de l'exécution de certains aspects en fonctions du contexte, de règles de dépendances et d'incompatibilités inter-aspects. Cependant, la politique de gestion de composition est définie par l'intégrateur et enrichie par l'adaptateur. Ainsi, aucun processus de validation de politique de composition n'est fourni par le framework, ce qui rend la tâche de composition informelle et donc source d'erreurs. Même si elle est statique, la composition des comportements dans le framework Molène demeure également une tache délicate et source d'erreurs car elle est gérée de manière ad-hoc par l'intégrateur des politiques d'auto-adaptation au niveau des automates.

Le mécanisme de fusion de Noah consiste à composer un ensemble de règles d'interactions portant sur une même fonctionnalité d'un composant. La fusion est basée sur les opérateurs du langage ISL et sur un ensemble de règles de compatibilité entre ces opérateurs. L'opération de fusion est commutative et associative. Ainsi, appliquer un ensemble de règles d'interactions sur un composant conduit toujours à un résultat équivalent en terme de comportement quelque soit l'ordre dans lequel on applique les règles. Ceci peut être considéré comme une forme de vérification automatisée des compositions effectuées. Le principe de fusion permet un entrelacement des adaptations et représente donc une amélioration par rapport aux approches précédentes.

D'autres travaux récents tentent de fournir une forme de vérification plus sophistiquée des compositions effectuées. Tessier et al. [117] se placent au niveau des modèles pour détecter les conflits entre les aspects. L'approche se limite pour le moment à la détection des conflits directs.

Douence et al. [34] utilisent un formalisme abstrait pour exprimer l'impact des aspects sur l'application de base. Le type de conflit traité ne concerne que les aspects avec état (pour cela des variables d'aspects sont définies), c'est à dire lorsque deux aspects ou plus partagent une même variable. Des règles de composition permettent ensuite d'identifier les coupes qui se chevauchent et les advices qui interfèrent entre eux.

Le projet SECRET (Semantic Reasoning Tool) [35] vise à améliorer le mécanisme de composition des filtres de composition. Le principe consiste à analyser l'impact[2] de chaque type de filtre sur les ressources qu'il utilisé (état de la condition, cible du message, appellant, nom du message, paramètres, deadline du message, temps). SECRET s'intéresse plus à l'impact de la composition sur une application donnée que l'impact de la composition des filtres entre eux[3]. Le comportement de tous les types de filtre est connu à l'exception près du type de filtre Meta dont le comportement est, par nature, indéfini et non prédictible. Les filtres de type Meta ne peuvent donc pas être pris en compte dans cette vérification de composition puisque leur impact n'est pas formalisable. Par ailleurs, bien qu'un formalisme soit utilisé pour décrire la sémantique associée aux autres filtres, aucune preuve formelle des règles de composition n'est fournie. Seul le prototype implémenté au dessus de Compose* fait acte de validation de l'approche.

Enfin, la notation CompAr [96] permet de spécifier les contraintes liées à l'application de plusieurs aspects comportant des advices de type around. Un compilateur est chargé de vérifier la conformité avec les contraintes afin de détecter les conflits. Cette approche est très proche de celle de SECRET du point de vue du formalisme utilisé. Par contre, elle se base sur l'annotation des aspects par des contraintes spécifiées par les concepteurs ou développeurs contrairement à celle proposée par SECRET qui ne se base que sur la sémantique des types de filtre et des opérations sur les ressources.

Problème des cycles

Définition du problème. Le remplacement d'un composant a pour effet de modifier la topologie des assemblages et peut donc entraîner des apparitions de cycles c'est à dire des boucles infinies dans un flot d'appels de méthodes. Un cycle est malin s'il engendre une boucle infinie. Par exemple, si Anne-Marie notifie ses rendez-vous à l'agenda de Michel qui notifie lui-même ses propres rendez-vous à l'agenda de Mireille qui notifie les siens à l'agenda de Anne-Marie, dans ce cas, on a une boucle infinie.

Les règles de typage sur les connexions utilisées en Fractal ou en CCM assurent que les modifications d'assemblage sont valides mais ces deux modèles ne vérifient pas que la modification des assemblages de composants (par ajout ou remplacement de composants) introduit des cycles. Grâce aux protocoles, SOFA détecte les cycles (divergence). Dans les approches orientées aspect telles que JAC, Noah, les filtres de composition et le modèle FAC, l'utilisation de coupes génériques (utilisant le « * » par exemple) peut aussi entraîner l'apparition de cycles non détectés.

Problème des points de non-déterminisme

Définition du problème. Un point de non-déterminisme correspond à un point dans le graphe d'interactions où les fonctionnalités d'une entité logicielle peuvent potentiellement être appelées dans n'importe quel ordre. Les points de non-déterminisme sont engendrés par des actions non ordonnées qui font intervenir des entités logicielles communes. L'ordre d'exécution de ces actions peut varier à chaque fois. De ce fait, suivant la nature de l'entité logicielle, le point de point de non-déterminisme peut engendrer des conflits. En effet, si une entité logicielle « partagée » par

[2] L'impact d'un filtre est actuellement représenté par les opérations en lecture et en écriture sur les ressources.
[3] Seuls les conflits impliquant une réduction des fonctionnalités de l'application superimposée sont considérés.

ces actions impose un mode d'utilisation particulier, on ne peut pas être certain qu'elle sera correctement utilisée.

Un exemple de point de non-déterminisme dans un assemblage de composants est donné dans la figure 3.1. Supposons que, dans un même flot d'exécution de méthodes, le composant *a* utilise deux composants *b* et *c* (dans une même connexion), et *b* et *c* utilisent tous les deux un même composant *d*. Dans ce cas, à partir de *a*, les fonctionnalités de *d* peuvent potentiellement être appelées dans n'importe quel ordre ce qui pose problème si *d* est un buffer avec les fonctionnalités de lecture et d'écriture qui ne sont pas interchangeables en terme d'ordre d'utilisation.

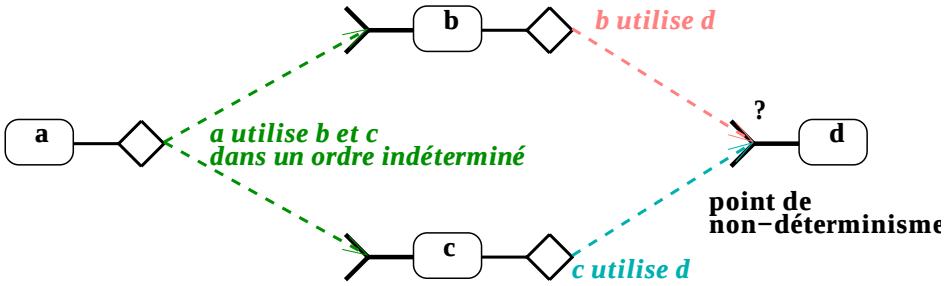

FIG. 3.1 – Point de non-déterminisme dans l'exemple du buffer

Dans les approches orientées aspect, des points de non-déterminisme peuvent apparaître lorsque le comportement d'une fonctionnalité est modifié plusieurs fois et que la composition de ces comportements n'impose pas d'ordre entre les différentes « actions » à effectuer. Dans ce cas, si certaines de ces actions font intervenir une même entité logicielle, un point de non-déterminisme est introduit car on ne peut pas prédire quel sera l'ordre d'exécution des actions choisi par le processeur.

Même si les problèmes de compositions comportementales sont généralement associés aux approches orientées aspects, les problèmes dus aux points de non-déterminisme se posent également dans les approches à composants. En effet, certains types de liaison permettent de « lier » un composant à plusieurs autres composants simultanément. La manière dont les composants liés par la connexion sont appelés dépend des implémentations. Les appels à chaque composant peuvent être séquentialisés ou bien un message peut être diffusé en même temps à tous les composants (dans le modèle CCM, par exemple, plusieurs puits peuvent être connectés sur la même source et interagir par diffusion de manière asynchrone). Dans ce dernier cas, il peut y avoir de point de non-déterminisme à l'intérieur d'un assemblage de composants puisqu'on ne peut pas savoir à l'avance quel sera l'ordre d'exécution des actions choisi par le processeur.

A notre connaissance, aucune des plates-formes à aspects et à composants étudiées ne traite ce type d'erreurs. Noah est concerné car propose un opérateur de fusion non ordonné. De même pour les modèles à composants CCM (ports multiples de type *EventSouce* et ports simples de type *EventSink*) et Sofa (connecteurs *EventDelivery* et *EventChannelDelivery*) qui permettent les liaisons $1 - n$ entre composants sans ordonnancement. Pour les autres approches, il n'existe pas de type d'opérateur ou de connexion explicite pour effectuer des actions de manière non ordonnée. Cependant, l'indéterminisme peut être introduit d'une autre manière. Par exemple, dans JAC, si l'aspect de composition ne définit aucune politique de composition (aucune règle de dépendances et d'incompatibilités inter-aspects), cela revient à dire que l'on n'impose pas d'ordre entre les actions des aspects tissés sur une même coupe et que des points de non-déterminisme peuvent apparaître. Les filtres de composition, Fractal, OSGi, Molène et FAC ne semblent pas être directement concernés par ce type d'erreurs.

Problème de la non anticipation des erreurs liées à une adaptation

Définition du problème. Les modifications induites par une adaptation donnée sont très souvent effectuées par programmation au lieu d'être explicitement décrites. Cela rend la tache de comprendre ce que fait une adaptation plus difficile. De plus, les erreurs de typage dynamique, de consistance des assemblages, de cycle et de cohérence de composition que ces modifications peuvent entraîner ne peuvent pas être anticipées. En effet, pour les cas d'erreurs gérés par la plate-forme, l'adaptation ne peut être invalidée qu'en rattrapant les exceptions induites par des appels à l'API d'adaptation de la plate-forme ce qui réduit les impacts d'une mauvaise adaptation. Cependant, il faut être capable de revenir en arrière. Pour les erreurs non prévues par la plate-forme, une mauvaise adaptation des composants ne lève pas d'exceptions et il faut rattraper les erreurs au moment où elles surviennent durant l'exécution de l'application avec le risque de ne pouvoir les identifier et donc les traiter ou de traiter seulement les symptômes mais pas la source de l'erreur elle-même.

Par exemple, dans le modèle CCM, les opérations de connexion entre composants (`connect_XXX`) peuvent lever les exceptions `AlreadyConnected`, `InvalidConnection` et `ExceededConnectionLimit`. Quant aux opérations de deconnexions (`disconnect_XXX`), elles peuvent lever les exceptions `NoConnection` et `InvalidConnection`. Dans Fractal/Julia, les opérations d'adaptation telles que `addFcSubComponent`, `bindFc`/`unbindFc` peuvent lever les exceptions `NoSuchInterfaceException`, `IllegalBindingException`, `IllegalLifeCycleException` ou encore `IllegalArgumentException`.

Par contre, dans ces modèles, les cycles ne sont pas gérés et représentent un exemple d'erreurs non prévues. A l'exécution, l'erreur finit par se manifester par une exception de type `StackOverFlowException` par exemple, ce qui ne permet pas de traiter l'erreur correctement car l'exception n'est pas liée à l'erreur mais à son impact sur le système.

Problème du moment d'adaptation

Définition du problème. Le problème de savoir quand il est sûr de procéder à une adaptation a été soulevé par [94, 33, 16, 8]. Il s'agit de déterminer quand les modifications liées à une adaptation peuvent être prises en compte dans l'exécution de l'application. Alors que la question semble primordiale, des travaux étudiés au chapitre 2, seuls Molène, Fractal et SOFA mettent en œuvre des mécanismes pour y répondre.

Molène implémente un mécanisme permettant de forcer l'attente de terminaison de toutes les requêtes en cours de traitement avant d'adapter les composants. Ceci n'est pas toujours approprié (une requête peut ne jamais obtenir de réponse), en particulier dans le contexte d'applications distribuées et mobiles qui sont spécialement visées par le framework.

Un contrôleur Fractal particulier (`LifeCycleController`) gère le cycle de vie du composant (et des sous composants) auquel il est associé. Il permet d'arrêter le composant à adapter et de le remettre en marche quand les modifications nécessaires ont été effectuées. Ainsi lorsqu'on modifie le contenu d'un composant (ajouter/retirer un composant ou changer une connexion à un certain niveau), il est nécessaire de stopper tout le composant englobant le niveau en question et dans certains cas, ses clients. Ceci est très coûteux, et peut impliquer l'arrêt d'une partie non négligeable de l'application, si le changement doit impliquer de remonter et d'intervenir à un très haut niveau de la hiérarchie des composants. Pendant le temps d'adaptation, une partie entière de l'application ne peut plus être utilisée. L'ensemble des composants qui doivent être arrêtés n'est pas calculé automatiquement. C'est à l'adaptateur d'arrêter manuellement, via le contrôleur, les composants qui sont censés être concernés par les modifications.

Une caractéristique de SOFA est que la spécification des protocoles de composants permet de fixer des moments potentiels d'adaptation pour indiquer les moments précis où une adaptation peut avoir lieu. Un moment d'adaptation est décrit par un point de début et un point de fin et ces points doivent être placés de façon à ce qu'aucun événement soit intercalé entre le point de début et celui de fin de l'adaptation. Ainsi, les composants SOFA n'ont pas besoin d'être arrêtés pour être adaptés contrairement aux composants Molène et Fractal. Néanmoins, le développeur a la charge de définir lui-même les bons moments d'adaptation dans les protocoles. Or, aucun outil n'est fourni pour vérifier que les points d'adaptation choisis sont corrects. De plus, un moment d'adaptation ne correspond pas à un type d'adaptation particulier : les moments acceptant une adaptation sont les mêmes pour toutes les adaptations quelque soit leur type.

3.1.2 Synthèse

Pour assurer que le service fourni ne diverge du service attendu à cause d'une adaptation, les systèmes adaptatifs dynamiques prennent en charge la vérification de certaines classes d'erreurs induites par adaptation. La gestion de la consistance des assemblages prime pour les plates-formes à composants et la prise en charge de la cohérence de composition des adaptations (i.e. les aspects) est essentielle pour les approches par aspects. Pourtant, en plus des types d'adaptations mis en œuvre explicitement, il existe des adaptations cachées. Par exemple, la modification comportementale d'aspects peut entraîner une modification du graphe d'interactions entre les différentes entités logicielles et l'altération des assemblages de composants peut provoquer une modification du comportement des composants.

Le tableau 3.1 met en évidence le fait que les points forts en terme de vérification des approches par composants font partie des points faibles des approches par aspects et vice-versa alors qu'un certain nombre de vérifications sont nécessaires et communes aux deux approches. D'autre part, les vérifications effectuées changent d'une plate-forme à une autre en fonction de la mise en œuvre des adaptations. Elles ne peuvent donc pas être réutilisées. Enfin, ces vérifications ne sont pas systématiques, et les techniques de vérification employées sont souvent à la charge du développeur, intégrateur ou adaptateur.

		Vérifications effectuées à l'adaptation	Problèmes introduits à l'exécution par adaptation
Approches à aspects	JAC	Cohérence de composition des adaptations (partielle)	Typage dynamique, cycles, points de non-déterminisme
	Composition Filters		Typage dynamique, incohérence de composition, cycles
	Noah	Cohérence de composition des adaptations	Typage dynamique, cycles, points de non-déterminisme
Approches à composants	CCM / OpenCCM	Consistance des assemblages (partielle)	Cycles, points de non-déterminisme
	Fractal / Julia	Consistance des assemblages	Incohérence de composition (statique), cycles
	Sofa / DCUP	Consistance des assemblages, cycles	Points de non-déterminisme
	Molène		Incohérence de composition (statique), cycles
	OSGi	Consistance des assemblages	Cycles
Approches mixtes	FAC / Julius	Consistance des assemblages	Incohérence de composition, cycles

TAB. 3.1 – Types d'erreur et prises en charge par les plates-formes

Les modifications associées à une adaptation sont souvent effectuées de manière programmative [89, 86, 101, 100]. Par conséquent, ces plates-formes ne peuvent détecter des erreurs que lorsque les modifications ont été effectuées. Ceci rend la gestion d'erreurs plus difficile puisqu'il faut « réparer » au lieu de simplement prévenir. Un simple retour en arrière n'est pas toujours suffisant : plus l'erreur tarde à apparaître plus il est difficile de la rattraper. D'autre part, certaines approches essayent de déterminer quand il est sûr d'adapter les composants [109, 86, 101]. Cependant les techniques employées présentent des inconvénients : soit elles dégradent les performances du système soit elles ne sont pas assez flexibles.

Les enjeux en terme de vérifications en environnements dynamiques connus, nous pouvons évaluer quelles approches de vérifications statiques de programmes peuvent être réutilisées dans le cadre de la problématique de sûreté des adaptations dynamiques et dans quelle mesure.

3.2 Apport des approches statiques à la sûreté

Nous présentons dans cette section quatre domaines de l'informatique adressant la problématique de vérification des programme sous une forme particulière : le typage, la composition de comportements, le model-checking et les langages de description d'architecture. Ces domaines sont présentés en particulier car ils traitent de classes d'erreurs que nous avons étudiées dans la section précédente et qui surviennent aussi lorsqu'on adapte une application à l'exécution. Une brève description des mécanismes mis en œuvre dans ce domaine vont nous permettre d'évaluer dans quelle mesure ces approches peuvent être réutilisées dans le cadre de la problématique de sûreté en environnement dynamique.

3.2.1 Vérification du typage

Le fait qu'un programme « passe » la compilation n'assure pas qu'il fonctionne correctement. Il assure seulement que le programme ne contient pas certaines classes d'erreurs : erreurs syntaxiques (le compilateur refuse tout texte qui viole la grammaire), détection de code mort, violation de visibilité, erreurs de type. Dans la suite de cette section, nous nous focalisons sur la vérification du typage.

Les types représentent la forme la plus répandue de spécification des programmes. Ils documentent l'intention des développeurs aussi bien qu'ils permettent aux compilateurs de détecter et d'empêcher des erreurs telle que « message not understood » (la cible ne sait pas répondre au message qu'on lui envoie, c'est à dire qu'il n'existe pas dans son type de fonctionnalité similaire), « wrong argument type » (le type d'un objet passé en paramètre n'est pas conforme au type attendu de l'argument) et « bad assignment type » (le type de l'objet affecté n'est pas conforme au type de l'objet à affecter).

Dans les langages à objets tels que Java ou C++, le type d'un objet de classe A peut être assimilé à l'intension de la classe [25], c'est à dire l'ensemble des propriétés de A. Ainsi, on dit qu'un objet est de type t s'il vérifie toutes les propriétés de t. Le typage « objet » sert de base à la vérification de la correction des programmes et à la substitution des objets. Il permet d'assurer que les objets, de par la définition statique de leur type, sauront répondre aux messages prévus. L'élément pivot sur lequel se base le principe de typage est la définition d'une relation de substituabilité entre types d'objets permettant de les comparer et donnant la possibilité pour un objet d'un certain type t d'être utilisé comme un objet d'un autre type t'. La forme de substituabilité la plus forte en terme de sûreté est le sous-typage contravariant sur les paramètres [37] et covariant sur le type de retour.

Dans la suite, on note $o : t$ pour indiquer que l'objet o est de type t et $f : t \rightarrow s$ pour indiquer que la fonction f est de type fonctionnel $t \rightarrow s$. Les définitions suivantes ont été reprises de [27].

Soient τ un type contenant les propriétés p_1 de type fonctionnel $\alpha_1 \rightarrow \beta_1$ à p_n de type fonctionnel $\alpha_n \rightarrow \beta_n$ et τ' un type contenant les propriétés p'_1 de type fonctionnel $\delta_1 \rightarrow \gamma_1$ à p'_m de type fonctionnel $\delta_m \rightarrow \gamma_m$ avec $n \leq m$. On dit que τ' est un sous-type de τ, noté $\tau' \leq \tau$, si et seulement si pour tout $p_i : \alpha_i \rightarrow \beta_i$, il existe $p'_j : \delta_j \rightarrow \gamma_j$ tel que $(\delta_j \rightarrow \gamma_j) \leq (\alpha_i \rightarrow \beta_i)$ avec $i \in \{1, ..., n\}$ et $j \in \{1, ..., m\}$.

$\delta \rightarrow \gamma$ est un sous-type fonctionnel de $\alpha \rightarrow \beta$, noté $(\delta \rightarrow \gamma) \leq (\alpha \rightarrow \beta)$, si et seulement si $\alpha \leq \delta$ et $\gamma \leq \beta$. La relation $\alpha \leq \delta$ est appelée *contravariance* et la relation $\gamma \leq \beta$ est appelée *covariance*. Par exemple, soit $f : \alpha \rightarrow \beta$ et $a : \tau$. Si $\tau \leq \alpha$ alors $f(a)$ est définie. Intuitivement, une fonction f qui attend un argument de type α peut recevoir sans danger un argument d'un sous-type τ de α.

Evaluation

Si nous considérons que le type d'un composant correspond à l'ensemble de fonctionnalités qu'il offre, alors nous pouvons comparer l'ajout et le retrait de fonctionnalités dynamique à un changement du type du composant pendant son cycle de vie. Cependant, les travaux sur le typage, tel qu'il est défini dans les applications orientées objet [25, 37] par exemple, ne sont pas suffisants pour valider des évolutions de type pendant l'exécution. En effet, il n'est pas raisonnable d'envisager que les vérifications de types effectuées par le compilateur soient effectuées à nouveau à chaque changement de type durant l'exécution pour des raisons de performances. Quant aux langages actuels supportant la modification des interfaces à l'exécution, tels que Smalltalk, ils ne sont pas typés et détectent les erreurs seulement lorsque la fonctionnalité manquante est appelée. Le problème de la construction de types dynamiques et leur vérification, étudié en 3.1.1, reste donc un problème dont il n'existe pas d'implémentations satisfaisantes.

3.2.2 Composition de comportements

Etant donnés deux comportements C_1 et C_2, comment et sous quelles conditions composer les deux pour former un nouveau comportement C_3. La problématique de la composition est rencontrée dans différents domaines et traitée de diverses manières. Cependant, comme nous allons le constater dans la suite, l'élément commun à toutes ces approches est la notion d'ordre de composition.

Fonctions mathématiques

En mathématique, la composée de fonctions h, notée fog, consiste à appliquer g à x puis à appliquer f au résultat précédent, ce qui donne $h(x) = f(g(x))$. Cette sémantique de composition impose que x se trouve dans le domaine de définition de g, et que $g(x)$ soit dans le domaine de définition de f. L'ordre de composition a beaucoup d'importance : fog et gof ne donnent généralement pas le même résultat et n'ont pas les mêmes contraintes.

Notons que la même forme de composition est également présente dans les langages fonctionnels tels que Scheme [30] qui propose l'opérateur « compose » et la continuation de calcul. Interconnecter le point d'entrée de f avec la sortie de g impose des contraintes sur le nombre (f doit être une fonction unaire) et le type de leurs arguments.

Programmation orientée objet

Les langages de programmation orientés objet utilisent deux concepts de composition : l'héritage et l'agrégation.

L'héritage est un mécanisme de modification structurel et comportemental. Il peut être considéré comme une sorte de composition dans la mesure où on va chercher à combiner un ensemble de classes $\mathcal{C}$ avec un ensemble de modifications M pour obtenir une sous-classe SC imposant des contraintes sur la conception des sous-classes afin d'assurer une compatibilité à la fois structurelle et comportementale. La compatibilité structurelle est assurée par le sous-typage (via la relation d'héritage) que nous avons vu dans la section précédente. Quant, à la compatibilité comportementale, elle se pose dans trois situations :

- En cas d'héritage multiple (i.e. : l'ensemble $\mathcal{C}$ n'est pas réduit à un élément) avec propriétés communes, c'est à dire lorsque, par exemple, C_1 et C_2 définissent une méthode m et que C_3 hérite à la fois de C_1 et C_2.
- En cas de redéfinition de méthodes, la sous-classe définit une méthode m qui est déjà présente dans au moins une des classes de $\mathcal{C}$. Dans ce cas, la sous-classe doit indiquer quand et dans quel ordre appeler ses super classes.
- Lors d'un appel à une méthode non présente dans la classe effective de l'objet, la méthode la plus proche est recherchée dans la hiérarchie de classes dont l'objet est issu. Ce parcours de hiérarchie inclut encore une fois la notion d'ordre.

L'agrégation ou délégation consiste, au sein d'une classe, à utiliser d'autres classes. Pour accomplir une tâche t, un objet de classe C_1 utilise des objets de classe C_i à C_j où la séquence $\{i..j\}$ donne le comportement attendu. La composition des comportements des objets utilisés est donc définie par un séquencement des appels de messages sur ces objets.

Méta programmation

La méta programmation permet de modifier la sémantique d'interprétation d'un programme. Un langage de méta programmation sépare une application en deux niveaux : le niveau de base qui correspond à la partie fonctionnelle de l'application et un niveau méta qui correspond à la manière d'interpréter le niveau de base.

Le niveau de base est donc implémenté par les classes de l'application et le niveau méta est décrit dans des métaclasses. Les métaclasses s'instancient en métaobjets qui contrôlent les objets du niveau de base auxquels ils sont associés. La modification du comportement du niveau de base correspond donc, pour chaque objet du niveau de base, à désigner les métaobjets souhaités et les points où ces derniers interviennent. Le problème de la composition intervient donc dans cette approche lorsque deux métaobjets agissent sur un objet en un même point. Certains travaux se sont intéressés à la mise en œuvre de cette composition par la coopération de métaobjets [80] ou encore par héritage de métaclasses [18], c'est-à-dire qu'elles reposent sur les moyens de composition orientés objets (l'héritage et l'agrégation vues précédemment) : l'ordonnancement.

Programmation orientée sujet

La programmation orientée sujet (SOP, Subject Oriented Programming) [95] part de la constatation qu'il est très difficile de faire développer des applications par plusieurs équipes, d'étendre de manière non planifiée les applications, ou encore d'intégrer de manière non planifiée des applications qui ont été développées séparément.

Pour résoudre ce problème, la programmation orientée sujet étend la programmation orientée objet en lui ajoutant des capacités de décomposition additionnelles. Elle permet de capturer différentes vues (les sujets) d'une seule classe, c'est à dire les différents rôles que peut jouer un objet. Programmer par sujets pour réaliser, étendre ou intégrer des applications consiste alors à déterminer et à écrire les sujets représentant les différents points de vues sur l'application puis à réunir les différentes vues en utilisant un langage de composition pour obtenir l'application finale.

Le mécanisme de composition de ces différentes vues est spécifié par des relations de composition et se fait manuellement. Une relation de composition identifie les éléments communs aux différentes vues prises deux à deux et spécifie comment les éléments correspondants doivent être composés. Plusieurs règles de composition sont possibles : l'écrasement (une propriété est prépondérante), ou la fusion (il faut alors fondre deux propriétés en une seule). Dans le dernier cas, lorsque les deux propriétés à fusionner sont des méthodes de corps respectif C_1 et C_2 alors le corps de la méthode résultante est soit C_1 suivi de C_2 en séquence ou inversement. On en revient encore à un séquencement des appels de messages.

Evaluation

Les mécanismes de composition vus précédemment sont tous basés sur un ordre de composition. Un grand nombre d'approches orientées aspects dont celles que nous avons vu au chapitre 2 s'appuient sur la méta programmation[4]. La composition de métaobjets servant souvent de support pour implémenter la composition d'aspects, l'ordonnancement devient alors le mécanisme sous-jacent de composition implicite d'un certain nombre d'approches orientées aspects. Or, tous les ordres de composition ne sont généralement pas équivalents et un certain nombre de conflits de composition peuvent survenir quelque soit l'ordre de composition choisi [19, 97, 49]. Les mécanismes de composition proposés par les approches statiques n'apportent donc pas d'éléments de réponse au problème de la cohérence de composition d'adaptations comportementales des approches dynamiques étudié en 3.1.1.

3.2.3 Model-checking

Principe

Le model-checking (vérification de modèles) [119, 66, 29] est une technique de vérification formelle qui repose sur une idée simple : si on énumère toutes les situations possibles auxquelles peut mener le programme, on sait s'assurer qu'aucune de ces situations n'est en contradiction avec les comportements désirés sous l'hypothèse qu'il n'y a pas de panne.

Le model-checking est basé sur deux modèles : un modèle du système représentant tous les états possibles du programme à vérifier et un modèle des propriétés que ce dernier est censé préserver. Les modèles de système utilisés par les model-checkers sont généralement des modèles de type états / transitions tels que les automates ou graphes de Büchi, tandis que les modèles des propriétés sont généralement des formules de logique temporelle telles que CTL (Computational Tree Logic) qui est une logique des chemins ou PLTL (Propositional Linear Temporal Logic) qui est une logique d'état.

La technique de vérification consiste alors à prouver la cohérence de deux modèles : est ce que le système d'états / transitions vérifie une formule de logique temporelle donnée ? Les propriétés exprimables en logique temporelle peuvent être classées en cinq grandes catégories : sûreté, vivacité, atteignabilité, équité et absence de blocage. Différents algorithmes ont été proposés suivant le modèle des propriétés utilisé. Pour les propriétés exprimables en CTL, l'algorithme de base est un algorithme de marquage des états du système par les propriétés qui le satisfont. Pour les propriétés exprimables en PLTL, l'algorithme de base consiste à vérifier que l'intersection de l'ensemble des mots définis par le graphe / automate représentant le système et du langage défini par la négation de la propriété est vide.

[4] Dans ce cas, les fonctionnalités de l'application sont implémentées au niveau de base, et les aspects sont implémentés sous forme de métaobjets.

Le model-checking a pour avantage de fournir en général un contre exemple lorsqu'il détecte la violation d'une propriété par le système. Ceci facilite grandement la compréhension et la correction des erreurs. D'autre part, il ne requiert aucune interaction avec l'utilisateur, qui soumet à la fois les modèles du système et des propriétés à un model-checker, et qui attend que celui-ci ait terminé l'examen de toutes les situations possibles. Néanmoins, la construction des modèles reste une tâche difficile et incontournable qui incombe à l'utilisateur.

Evaluation

L'une des caractéristiques des systèmes adaptatifs dynamiques est la non anticipation des évolutions futures. Or, le model-checking n'adresse que les problèmes où le nombre d'états est fini. C'est à dire qu'il nécessite la représentation de toutes les situations possibles[5] et ne peut donc pas être appliqué directement aux systèmes adaptatifs dynamiques. Cependant, il est possible de s'inspirer de certaines propriétés de logique temporelle du model-checking notamment pour détecter les cycles et les points de non-déterminisme, problèmes étudiés en 3.1.1, dans les réseaux d'entités logicielles coopérantes (sous forme d'assemblage explicite ou non) pouvant être assimilés à des graphes.

3.2.4 Langages de description d'architecture

Principe

Les langages de description d'architecture (ADL) [76] sont des notations destinées à représenter l'architecture d'un système logiciel en vue de son analyse. L'utilisation des ADLs se situe principalement au moment de la conception d'un système.

Dans les ADLs, l'architecture d'un système est décrite principalement en terme de *composants* qui implémentent des *interfaces*, de *connecteurs* (interconnexions entre composants) et de leurs *configurations* (ou compositions). Un composant est présenté de manière grossière dans le cadre des ADLs comme une unité de calcul ou un entrepôt de données. Une interface spécifie les services que le composant fournit. Les connecteurs modélisent les interactions entre les composants à travers leurs interfaces ainsi que les règles qui gouvernent ces interactions. Une configuration (ou composition) représente un graphe de composants connectés entre eux à l'aide de connecteurs.

Les ADLs sont généralement accompagnés par une panoplie d'outils permettant la modélisation contrôlée des architectures (en limitant les possibilités d'actions en fonction de l'état courant de la modélisation de l'architecture[6]), l'analyse de l'architecture (model checkers, parsers, ...), la génération de code, la simulation de l'architecture.

La définition d'interfaces répond de manière très élémentaire à l'expression de la sémantique des composants. Pour permettre d'utiliser les outils décrits précédemment (vérification de contraintes, simulation des architectures), il est nécessaire de disposer d'un modèle définissant la sémantique des composants. Par exemple, Rapide repose sur l'ordonnancement partiel d'événements, et définit la sémantique comportementale [68]. Wright utilise le formalisme CSP pour analyser les connexions entre connecteurs et composants afin de détecter des deadlocks [6].

La majorité des ADLs existants ne s'intéressent qu'aux configurations statiques. Les exceptions sont C2SADEL [75], Darwin [69], Rapide [68], et Weaves [46]. Darwin et Rapide supportent seulement les modifications dynamiques d'architecture qui sont connues à l'avance. C2SADEL et Weaves permettent l'ajout, la suppression et le ré-assemblage de composants.

[5] A noter que cette obligation d'exhaustivité conduit rapidement à un dépassement des capacités en mémoire. Ce phénomène est connu sous le nom d'explosion combinatoire, et la recherche de solutions à ce problème est un domaine important de l'informatique.

[6] Par exemple, la sélection de composants dont les interfaces ne sont pas couramment utilisées dans l'architecture peut être interdite.

Evaluation

Les ADLs sont exploités pendant la phase de conception des applications et sont rarement disponibles pendant la phase d'exécution de celles-ci. Aussi, sont-ils insuffisants dans l'état actuel pour garantir que des changements dynamiques seront appliqués au système d'une façon sûre. La plupart des ADLs garantissent, en effet, la validité d'assemblage que pour les constructions initiales. D'après Medvidovic [76, 74], même les ADLs capables de modéliser des changements dynamiques (anticipés ou non) sont insuffisants pour garantir que les changements sont appliqués de manière sûre. Medvidovic précise que les problèmes de sûreté dus aux adaptations dynamiques tel que le problème de la consistance des assemblages, étudié en 3.1.1, n'entrent pas dans le cadre des ADLs et doivent être pris en charge par des outils d'analyse spécifiques aux environnements d'exécution.

Nous avons vu comment les adaptations dynamiques sont mises en œuvre dans différentes plates-formes. Nous avons identifié un certain nombre d'erreurs qui peuvent survenir lors d'adaptations dynamiques d'une application et comment les plates-formes gèrent ces erreurs. Le chapitre suivant synthétise les points importants abordés dans cet état de l'art et rebondit sur les objectifs de cette thèse.

« Pour un jour de synthèse, il faut des années d'analyse. »

Fustel de Coulanges

« Ce n'est pas assez de faire des pas qui doivent un jour conduire au but, chaque pas doit être lui-même un but en même temps qu'il nous porte en avant. »

Goethe

4

Synthèse et objectifs

DANS ce chapitre nous récapitulons les éléments appris des études présentées dans les deux chapitres précédents et en tirons un certain nombre d'objectifs.

Des types d'adaptations élémentaires et des mises en œuvre variées

De l'étude des canevas orientés aspects et des plates-formes à composants du chapitre 2, nous avons dégagé sept types d'adaptation élémentaires catégorisés en trois familles.

- **Evolution des interfaces** des entités logicielles : ajout et retrait de fonctionnalités, modification de la signature des fonctionnalités.
- **Composition comportementale** : modification répétée du comportement associé à une fonctionnalité.
- **Altération des assemblages** : ajout, retrait et remplacement d'entités logicielles.

Canevas orientés aspects et plates-formes ne prennent pas en charge les mêmes types d'adaptations élémentaires de manière explicite.

- Les adaptations de la famille altération des assemblages sont rendues explicites dans les approches par composants via une API dédiée.
- Les adaptations des familles composition comportementale et **évolution des interfaces** sont rendues explicites dans les approches orientées aspects par des mécanismes de tissage.

Les adaptations élémentaires ne sont pas mises en œuvre de la même manière par les plates-formes (ou canevas) d'une approche comme de l'autre. De plus, bien que certaines adaptations soient mises en œuvre explicitement, il existe des adaptations cachées.

- La modification comportementale engendre dans certains cas une modification du graphe d'interactions entre les différentes entités logicielles (i.e. : une altération des assemblages).
- L'altération des assemblages de composants engendre dans certains cas une modification du comportement des composants.

Des problèmes de sûreté en dynamique et leur prise en charge

Dans le chapitre 3, nous avons étudié quels types de problèmes de sûreté émergent en raison des adaptations dynamiques des applications et nous avons vu comment les plates-formes étudiées répondent à ces problèmes.

Canevas orientés aspects et plates-formes ne prenant pas en charge les mêmes types d'adaptations élémentaires de manière explicite, les vérifications effectuées ne sont pas du même ordre.

- Les plates-formes à composants se focalisent sur la **consistance des assemblages**.
- Les approches par aspects se concentrent sur la **cohérence de composition** des aspects.

En terme de vérifications, nous pouvons lister les points faibles suivants.

- Les types d'erreurs induits par des adaptations cachées ne sont pas pris en charge.
- Les différences de mise en œuvre des adaptations marquent des différences en terme de mise en œuvre des vérifications effectuées : les solutions sont donc spécifiques aux besoins de chaque plate-forme et ne peuvent pas être réutilisées dans d'autres plates-formes.
- Les vérifications effectuées ne sont pas systématiques, et les techniques de vérification employées sont souvent à la charge du développeur, intégrateur ou adaptateur.
- Les erreurs ne peuvent être anticipées lorsque les adaptations sont programmées.
- Les techniques utilisées pour déterminer quand il est sûr d'adapter les composants présentent des inconvénients majeurs (dégradant les performances du système ou insuffisamment flexibles).

Limites des approches statiques à la sûreté

En comparant les techniques destinées à fiabiliser la construction des applications aux besoins en terme de sûreté des systèmes adaptatifs dynamiques (plates-formes à composants et canevas orientés aspects), il ressort que les solutions existantes ne peuvent pas toujours être employées pour déterminer la sûreté des trois grandes familles de types d'adaptations élémentaires.

- **Evolution du type**. Est-il possible d'appeler une fonctionnalité inconnue ? L'ajout et le retrait de fonctionnalités dynamique correspondent à un changement du type du composant pendant son cycle de vie. Les travaux sur le typage [25, 37] interdisent les évolutions de type pendant l'exécution. Les langages actuels supportant la modification des interfaces à l'exécution ne sont pas typés et détectent les erreurs seulement lorsque la fonctionnalité manquante est appelée. La construction de types dynamiques et leur vérification reste donc un problème dont il existe pas d'implémentations satisfaisantes.

- **Composition comportementale**. Tous les ordres de composition ne sont généralement pas équivalents et un certain nombre de conflits de composition peuvent survenir quelque soit l'ordre de composition choisi [19, 97, 49]. Les approches basées sur l'ordonnancement [100, 13, 21] ne permettent donc pas de valider cette composition vis-à-vis des conflits potentiels. Les mécanismes de composition proposées par les approches statiques (composée de fonctions, héritage, délégation, ...) étant basés sur un ordre de composition, ils n'apportent donc pas d'éléments aidant à lever les conflits de composition des adaptations comportementales. Des travaux plus récents [96, 35, 117, 34] facilitent la détection de certaines classes de conflits mais restent néanmoins limités.
  - soit les conflits ne sont pas de détecter automatiquement.
  - soit la sémantique liée à chaque aspect doit être décrite par l'utilisateur.
  - soit les règles de composition ne sont pas formellement prouvées.

  Aucune solution proposée à ce jour ne fait l'unanimité auprès des chercheurs.

– **Altération des assemblages**. La connexion entre deux composants est-elle correcte syntaxiquement et sémantiquement ? Est-ce qu'il manque un composant dans un assemblage ? En terme de sûreté, les langages de description d'architectures [76] sont essentiellement exploités pour garantir la validité des assemblages initiaux.
  L'une des caractéristiques des systèmes adaptatifs dynamiques étant la non anticipation des évolutions futures, le model-checking [119, 66, 29] ne peut donc pas être appliqué directement aux systèmes adaptatifs dynamiques. Cependant, il est possible de s'inspirer de certaines propriétés de logique temporelle vérifiées dans la technique de model-checking notamment pour détecter les cycles et les points de non-déterminisme dans les réseaux d'entités logicielles coopérantes pouvant être assimilés à des graphes.

Objectifs

La sûreté de fonctionnement d'un système peut être assurée de trois façons : la prévention, l'élimination et la tolérance aux fautes. Notre approche pour la sûreté de fonctionnement en environnement dynamique se base sur la première solution : la **prévention** visant à empêcher l'**introduction de fautes**. Dans notre cas, nous supposons les applications sans faute de conception ni de fabrication et nous vérifions que les adaptations des composants durant l'exécution du système ne remettent pas en cause le bon fonctionnement de l'application.

Pour fiabiliser le processus d'adaptation, nous proposons d'identifier des **propriétés de sûreté** c'est à dire des critères qui permettent d'identifier les adaptations sûres (*Quoi*), de déterminer les moments d'adaptation sûrs (*Quand*) et de définir les caractéristiques d'une mise en œuvre sûre les adaptations (*Comment*). Les propriétés liées au *Quoi* et au *Quand* doivent être vérifiées dynamiquement mais avant qu'une adaptation ne soit effectivement réalisée pour éviter que l'adaptation ne mette le système dans un état incohérent et anticiper les problèmes. Les propriétés liées au *Comment* ne dépendent pas d'une adaptation particulière mais de la manière dont celles-ci sont mises en œuvre. Il est donc possible de vérifier statiquement si une plate-forme permettant les adaptations dynamiques prend bien en compte ou non ces propriétés.

Comme nous l'avons vu, les vérifications liées au contrôle d'une adaptation qu'effectuent les plates-formes sont spécifiques à chaque plate-forme bien qu'elles répondent à une problématique commune. Il n'y a donc pas de capitalisation des concepts de plus haut niveau ni de réutilisation possible de ces vérifications pour d'autres plates-formes ne maîtrisant pas les adaptations qu'elles permettent. L'idée clé de notre approche consiste alors à ne pas « coller » à un modèle de plates-formes particulier mais d'articuler les propriétés de sûreté au niveau d'un modèle abstrait de sûreté d'adaptations afin de fournir une capitalisation des connaissances dans le domaine de la sûreté des applications en présence d'adaptations dynamiques. Cette capitalisation doit permettre de mettre en œuvre les propriétés de sûreté à l'aide de solutions technologiques variées.

Dans notre approche pour empêcher l'introduction de fautes lors d'adaptations dynamiques, nous avons les objectifs suivants :

– Abstraire le concept d'adaptation pour fournir une solution indépendante des mises en œuvre des adaptations dans les plates-formes existantes ;
– Etendre la notion de type pour prendre en compte la gestion des évolutions structurelles et comportementales des entités logicielles adaptées dynamiquement ;
– Etablir une méthodologie pour formaliser et valider notre solution.

Dans la partie suivante nous présentons le cœur de notre approche pour la sûreté des adaptations. En nous servant de l'étude précédente, nous proposons une approche basée sur un modèle de sûreté d'adaptation indépendant des plates-formes à composants et des canevas orientés aspects. Ce modèle définit des propriétés de sûreté permettant de déterminer a priori si une adaptation va remettre en cause la sûreté de fonctionnement de l'application.

Deuxième partie

Un modèle de sûreté d'adaptation : Satin

« La supériorité de l'architecte le plus maladroit sur l'abeille la plus experte réside en ceci que l'architecte porte d'abord la maison dans sa tête »

Karl Marx

« La réflexion ne saurait être qu'une imitation, une reproduction du monde de l'intuition (...). Aussi peut-on dire très exactement que les concepts sont des représentations de représentations. »

Arthur Schopenhauer

5

Une architecture ouverte et paramétrable

La sûreté de fonctionnement d'un système peut être assurée de trois façons : la prévention, l'élimination et la tolérance aux fautes. Notre approche pour la sûreté de fonctionnement en environnement dynamique se base sur la première solution : la *prévention* visant à empêcher l'introduction de fautes. Pour détecter les problèmes, nous proposons d'identifier des *propriétés de sûreté* c'est à dire des critères qui permettent de déterminer des adaptations sûres. Ces propriétés doivent être vérifiées a priori, c'est à dire avant qu'une adaptation ne soit effectivement réalisée pour éviter que l'adaptation ne mette le système dans un état incohérent.

L'arrivée à maturité des applications distribuées a provoqué la multiplication des intergiciels (middleware). Or le coût pour passer d'une plate-forme à une autre est d'autant plus important que l'on ignore quelles seront les prochaines plates-formes à la mode. Le domaine de la sûreté des adaptations n'échappe pas à ce constat. Il n'y a pas de capitalisation des concepts de plus haut niveau ni de réutilisation possible des vérifications proposées par chaque plate-forme. La nécessité de concevoir des applications adaptables de façon sûre indépendamment de toute considération technique semble alors devenir essentielle.

L'Ingénierie Dirigée par les Modèles (IDM) est une forme d'ingénierie basée sur la construction générative d'applications informatiques à partir de modèles. L'IDM initie un passage d'une approche orientée « code » à une approche orientée « modèle ». Les notions de réutilisabilité, d'adaptabilité et d'évolutivité ne se situent alors plus au niveau du code mais au niveau du modèle.

L'idée clé de notre approche consiste à ne pas « coller » à un modèle de plates-formes particulier mais d'articuler les propriétés de sûreté au niveau d'un modèle abstrait de sûreté d'adaptations, nommé Satin (SAfety model for componenT adaptatIoNs) afin de fournir une capitalisation des connaissances dans le domaine de la sûreté des applications en présence d'adaptations dynamiques. Cette capitalisation doit permettre de mettre en œuvre les propriétés de sûreté à l'aide de solutions technologiques variées. Le modèle Satin est défini en UML. Les différents éléments de la spécification UML sont découverts au grès des besoins dans les chapitres de cette partie.

La section 5.1 présente les propriétés de sûreté étudiées en Satin. La section 5.2 décrit le modèle abstrait de sûreté d'adaptations. La section 5.3 donne un exemple permettant de comprendre le fonctionnement du modèle abstrait et des propriétés de sûreté. Enfin, la section 5.4 montre comment rendre le modèle abstrait opérationnel en s'appuyant sur l'IDM.

5.1 Propriétés de sûreté vérifiées en Satin

Comme nous l'avons vu dans la partie I, de la même façon qu'il est essentiel d'identifier si une adaptation est sûre (**propriétés du « Quoi »**), il est aussi indispensable, pour fiabiliser totalement le processus d'adaptation, d'identifier :

- à quels moments il est sûr d'adapter une applications (**propriétés du « Quand »**)
- comment les modifications induites par une adaptation doivent être effectuées (**propriétés du « Comment »**).

5.1.1 Propriétés du « Quoi » : sûreté liée à la nature des adaptations

La sûreté est un domaine large qui englobe les concepts de robustesse, disponibilité, fiabilité, confidentialité, et intégrité. Ainsi, on va plus s'intéresser à un concept qu'à un autre selon que l'on cherche à masquer complètement les fautes à l'utilisateur, à assurer une continuité de service en acceptant une dégradation temporaire de sa qualité, à sécuriser l'utilisation d'un service, ou à prévenir les conséquences les plus catastrophiques d'une défaillance en mettant le système dans un état sûr. Bien que ces notion soient fortement interdépendantes, nous écartons, vis-à-vis des adaptations dynamiques, les problèmes de confidentialité et d'intégrité qui ont trait à la sécurité ainsi que la disponibilité qui a trait à la qualité de service et qui sont traités par ailleurs [103, 45] pour nous concentrer sur les propriétés liées à la *fiabilité* d'exécution des applications. Ces propriétés visent à gérer les *types d'erreurs* étudiés en 3.1.1 et doivent être vérifiées a priori, c'est à dire avant qu'une adaptation ne soit effectivement réalisée pour éviter que l'adaptation ne mette le système dans un état incohérent.

Nous supposons les applications sans faute de conception ni de fabrication c'est à dire que les composants fournis initialement sont sûrs ainsi que les assemblages initiaux de ces composants. Ceci constitue notre hypothèse de départ et peut être considéré comme l'étape d'initialisation d'un raisonnement inductif sur la sûreté de l'application. Ensuite, nous vérifions qu'une adaptation de composants durant l'exécution qui fait passer l'application d'un état n à un état $n+1$ ne remet pas en cause le fonctionnement de l'application. Ainsi, nous avons identifié dix propriétés de sûreté plus ou moins importantes à vérifier suivant les plates-formes pour que les adaptations puissent être mises en œuvre :

- **P_0** : **Conservation du contexte d'utilisation**.
  Motivation : Eviter de substituer deux composants qui ne sont pas interchangeables en terme d'utilisation.
  Définition : Le composant remplaçant doit pouvoir être utilisé de la même manière que le composant remplacé.
  Applicabilité : Pour les plates-formes où le remplacement de composants est explicite.
  Précisions : Ne prend pas en compte la gestion du transfert d'état c'est à dire que l'état interne des données du composant remplaçant n'est a priori pas le même que celui du composant remplacé.
- **P_1** : **Consomption des messages**.
  Motivation : Eviter les erreurs dues à des appels à des fonctionnalités inconnues.
  - **P_{1a}** : **Adéquation des interfaces vis-à-vis de l'implantation**.
    Définition : Les interfaces, au sens des fonctionnalités accessibles d'un composant, doivent présenter uniquement des fonctionnalités auxquelles le composant sait effectivement répondre (c'est à dire des fonctionnalités qui sont implémentées d'une manière ou d'une autre).
  - **P_{1b}** : **Conservation des fonctionnalités de base**.
    Définition : Les fonctionnalités initiales d'un composant ne doivent pas être perdues au fil des adaptations.

Précisions : Les fonctionnalités initiales risquent d'être utilisées dans le code d'autres composants (ou du composant lui même) sans que cela n'apparaisse comme requis. Comme il est donc impossible de savoir si le retrait d'une fonctionnalité de base va rendre inutilisable d'autres composants (à moins d'analyser le code), il est préférable de garantir leur présence tout au long du cycle d'utilisation du composant.

- **P_2** : **Garantie de visibilité**.
  Motivation : Une adaptation ne doit pas requérir une certaine fonctionnalité d'un composant qui n'est pas visible depuis l'interface à laquelle l'adaptation a accès.
  Définition : Les fonctionnalités utilisées dans le cadre d'une adaptation doivent être visibles.
  Applicabilité : Pour les plates-formes où on ne peut s'adresser à un composant qu'à travers ses interfaces, et donc où la visibilité est partielle.

- **P_3** : **Consistance des assemblages**.
  Motivation : Pour qu'il n'y ait pas de composant requis manquant ou de type incorrect au sein d'un assemblage.
  Définition : Ce qui est utilisé par un composant dans le cadre d'une adaptation doit toujours être fourni.

- **P_4** : **Déterminisme du comportement**.
  Motivation : Pour éviter d'introduire de l'indéterminisme de manière involontaire. En particulier, on souhaite éviter que deux appels à une même fonctionnalité dans le même contexte (mêmes paramètres, mêmes états internes, mêmes ressources, mêmes adaptations, ...) conduisent à deux résultats différents.

  - **P_{4a}** : **Uniformité d'application d'une adaptation**.
    Définition : L'application d'une adaptation visant une modification comportementale doit impacter toutes les fonctionnalités correspondant à la coupe de l'adaptation.
    Applicabilité : Pour les plates-formes à composants incluant la notion de facettes, toutes les facettes comportant une fonctionnalité f doivent être impactées par l'application d'une adaptation dont la coupe vise f. Par contre, pour les plates-formes incluant la notion de point de vue, un composant peut fournir des fonctionnalités aux signatures équivalentes mais au comportement différent. Dans ce cas, la propriété d'uniformité ne s'applique pas : ces fonctionnalités peuvent donc être adaptées indépendamment.

  - **P_{4b}** : **Compatibilité des adaptations élémentaires**.
    Définition : Les adaptations élémentaires incompatibles ne doivent pas être appliquées à un même composant.
    Applicabilité : Pour les plates-formes permettant les adaptations élémentaires visant à modifier le comportement d'une fonctionnalité d'un composant (correspondant à la famille d'adaptation **composition comportementale**).

  - **P_{4c}** : **Points de non-déterminisme**.
    Définition : Il ne doit pas y avoir de point de non-déterminisme à l'intérieur d'un assemblage de composants.
    Applicabilité : Pour les plates-formes permettant d'utiliser plusieurs composants sans systématiquement imposer d'ordre d'enchaînement des appels à ces composants.

- **P_5** : **Cycles**.
  Motivation : Pour éviter des boucles infinies dans un flot d'exécution de méthodes.
  Définition : Il ne doit pas y avoir de cycle malin à l'intérieur d'un assemblage de composants.
  Précisions : Seuls les cycles résultant d'appels de méthodes introduits par adaptation peuvent être détectés. Quant à l'hypothèse de départ, elle garantit que le code initial de l'application ne contient pas de cycles.

- **P_6** : **Retro-activité des adaptations**.
  Motivation : Pour qu'une adaptation perdure dans le temps jusqu'à ce qu'elle soit retirée.
  Définition : Soit une adaptation élémentaire *ae* visant une modification comportementale dont la coupe est non close. L'ajout, ultérieur à l'application de *ae*, de fonctionnalités correspondant à la coupe de *ae* implique que l'adaptation *ae* soit applicable à ces nouvelles fonctionnalités[1].
  Applicabilité : Pour les plates-formes dont le temps d'adaptation est arborescent (i.e. : non linéaire).

Le tableau 5.1 indique pour chaque type d'adaptation élémentaire, les propriétés de sûreté qui doivent être préservées. Notons ici que la création et la destruction de composants sont également considérées comme des adaptations car ce sont des cas particulier d'ajout (connexion) et de retrait (déconnexion) de composants au sein d'un assemblage. Notons également que suivant la plate-forme visée et les types d'adaptation élémentaires que la plate-forme prend en charge, certaines propriétés de sûreté n'ont pas besoin d'être vérifiées soit parce qu'elles ne peuvent pas être mises en défaut (i.e. : elles sont toujours vraies), soit parce que leur vérification est prise en charge par la plate-forme, soit parce qu'il a été délibérément décidé de ne pas les vérifier (et de ne pas considérer leur violation comme une erreur).

Types d'adaptation	P_0	P_{1a}	P_{1b}	P_2	P_3	P_{4a}	P_{4b}	P_{4c}	P_5	P_6
Création de composants		x		x		x				
Connexion de composants				x	x				x	
Destruction / déconnexion de composants				x	x					
Remplacement de composant	x		x	x	x				x	
Ajout de fonctionnalité		x	x	x						x
Retrait de fonctionnalité			x		x					
Modification comportementale de fonctionnalité				x	x	x	x	x	x	

TAB. 5.1 – Synthèse sur les types d'adaptations élémentaires concernés par chaque propriété

Dans les propriétés qui sont à vérifier, il y a différents niveaux de précision dans la détection des erreurs. La violation des propriétés liées au typage (P_0, P_{1a}, P_{1b}, P_2 et P_3) va forcément provoquer une erreur à l'exécution avec comme symptôme d'appeler à un moment ou à un autre une fonctionnalité inconnue. La violation des propriétés induisant une modification comportementale (P_{4a}, P_{4b} et P_6) peut passer un peu plus inaperçue si la variation du comportement attendu n'est pas ressentie par l'utilisateur. La violation de ces propriétés ne provoque pas nécessairement d'erreur fatale à l'inverse de la violation des propriétés liées au typage provoquant l'arrêt impromptu de l'application, cependant, elle peut engendrer un comportement erroné. Enfin, la violation des propriétés globales (P_{4c} et P_5) est plus proche d'un « warning » que d'une erreur (au sens du compilateur) car certains points de non-déterminisme et certains cycles ne sont pas néfastes. Par exemple, l'utilisation de la récursivité est décelée comme un cas de cycle même si un cas d'arrêt correct de la récursivité est spécifié. Pour assurer que la violation d'une propriété globale ne conduit pas à une erreur nous sommes forcés de détecter tous les points de non-déterminisme et les cycles potentiels. Ensuite, une intervention humaine doit permettre de déterminer si le risque est présent ou non et de décider si finalement l'adaptation peut être réalisée ou non.

Les propriétés du « Quoi » peuvent être exprimées directement sur les éléments du modèle abstrait de Satin. Le chapitre 7 montre comment elles sont formalisées et validées.

[1] Celles-ci doivent être adaptées comme si elles étaient présentes lors de l'application de l'adaptation *ae*.

5.1.2 Propriétés du « Quand » et du « Comment » : sûreté liée au processus d'adaptation

A l'inverse des propriétés du « Quoi », les propriétés du « Quand » et du « Comment » ne sont pas liées à la nature des adaptations mais directement au processus d'adaptation. Or, le modèle Satin modélise les adaptations mais ne modélise pas le processus d'adaptation à proprement dit. Le processus d'adaptation est modélisé à la projection du modèle. La mise en œuvre d'un service de sûreté (chapitre 8) montre comment ces propriétés sont exprimées.

Propriétés du « Quand »

Après avoir déterminé qu'une adaptation est sûre, il faut déterminer quand est ce qu'il est sûr de procéder aux modifications liées à cette adaptation. Le problème du « quand adapter » de façon sûre se pose dans les termes suivants : à quel moment rendre « effective » une adaptation ? Pour simplifier le problème nous avons en effet choisi de dissocier le moment de la demande d'adaptation qui est toujours immédiate (acceptation ou rejet) et le moment où l'adaptation devient effective (le moment à partir duquel elle a un impact sur l'exécution des composants) qui lui dépend de l'état de l'application mais aussi de la nature des modifications à effectuer c'est à dire du type des adaptations élémentaires.

Evolution du type. Nous pouvons d'ors et déjà affirmer que l'ajout d'une nouvelle fonctionnalité ne perturbe pas l'exécution d'un composant et peut donc avoir lieu n'importe quand. Quant au retrait d'une fonctionnalité, il peut bien sûr avoir un impact sur l'exécution d'autres composants. Le moment où un retrait de fonctionnalité peut devenir effectif est donc guidé par le fait qu'aucun assemblage ne doit devenir inconsistant (propriété P_3 du Quoi). C'est à dire qu'une fonctionnalité ne peut être retirée tant qu'elle est utilisée par un autre composant au travers d'une adaptation (rappelons que seules les fonctionnalités ajoutées dynamiquement par adaptation peuvent être retirées). Notons qu'à ce niveau, nous n'attendons pas que la fonctionnalité ne soit plus utilisée pour accepter l'adaptation : nous rejetons directement la demande d'adaptation en signalant que la fonctionnalité est utilisée. De ce fait, les composants ne sont pas mis en attente et la demande d'adaptation peut être renouvelée et prise en compte lorsque les connexions de composants via cette fonctionnalité auront toutes été défaites.

Composition comportementale. La modification du comportement d'une fonctionnalité est délicate à prendre en compte. Tout dépend de la « nature » de la modification à effectuer. Tout d'abord, il est nécessaire de distinguer les modifications qui ont un effet de bord de celles qui n'en ont pas. Une modification comportementale qui n'a pas d'effet de bord sur les composants (tracer l'exécution d'un composant par exemple) peut être effectuée n'importe quand. D'autre part, il faut également prendre en compte le fait de ne pas interrompre un flot d'exécution contiguë c'est à dire une suite d'appels à des fonctionnalités qui démarque une transaction ou bien qui fait partie d'un contrat de synchronisation.

Altération des assemblages. L'ajout ou le remplacement d'un composant au sein d'un assemblage revient à une modification comportementale, on se retrouve dans le même cas qu'une adaptation de ce type. Comme pour le retrait d'une fonctionnalité, le moment du retrait d'un composant est guidé par le fait qu'aucun assemblage ne doit devenir inconsistant (propriété P_3 du Quoi).

Propriétés du « Comment »

La question de savoir comment adapter les composants de façon sûre est aussi importante que les deux questions précédentes. En effet, même si une adaptation ne viole pas les propriétés du Quoi et du Quand, elle peut toujours rendre une application inutilisable si elle n'a pas été mise en œuvre correctement. Par exemple, supposons que notre agenda soit connecté à une base de données et que l'on veuille gérer un mode déconnecté entre les deux. Cette adaptation implique

des modifications sur les deux composants. Or, si seul l'agenda est adapté parce qu'un problème s'est produit pendant le processus de mise en œuvre de l'adaptation, l'application peut se trouver dans un état incohérent. En effet, si l'agenda pose un verrou sur la base de données et se déconnecte momentanément, certains enregistrements de la base de données se trouvent inaccessibles (n'étant pas adaptée, la base de données ne sait pas comment réagir aux déconnexions).

Pour cette raison, le domaine des adaptations doit bénéficier des propriétés ACID au même titre que le domaine des bases de données. Nous rappelons ici qu'une adaptation peut être composée d'un ensemble d'adaptations élémentaires et que chaque adaptation élémentaire correspond à une modification structurelle ou comportementale d'un seul composant.

Atomicité. Cette propriété garantit que la mise en œuvre d'une adaptation suive le paradigme du « tout-ou-rien ». Si les modifications liées à une adaptation ne peuvent pas être toutes appliquées (l'une d'elles au moins viole une propriété du Quoi au moins ou une panne s'est produite), alors l'adaptation ne doit pas être appliquée du tout. L'atomicité assure que la restauration de l'état précédent de l'application (celui juste avant l'adaptation) est toujours possible.

Consistance. Ce critère assure qu'une adaptation laisse le système dans un état consistant une fois la mise en œuvre de l'adaptation terminée. Dans notre approche, les propriétés du Quoi représentent un ensemble de règles qui définit ce qu'est un état consistant. De plus, on suppose que 1) l'état initial du système est consistant, puisqu'il s'agit de l'état d'origine avant toute adaptation et 2) l'état résultant d'une adaptation précédente est consistant (par induction sur la propriété de consistance). Donc, une adaptation ne peut avoir lieu que sur un état consistant du système. Ensuite, durant la mise en œuvre de l'adaptation, certaines des propriétés du Quoi peuvent être violées (car toutes les modifications ne peuvent pas être effectuées au même moment). Mais une fois toutes les modifications effectuées, la consistance assure que le nouvel état est consistant. Comme, d'autre part, une adaptation est atomique, le système est consistant quoi qu'il arrive.

Isolation. Cette propriété empêche des adaptations concurrentes de voir les états intermédiaires potentiellement inconsistants les unes des autres. L'isolation permet à plusieurs adaptations de modifier le même ensemble de composants comme si chacune de ces adaptations était la seule à adapter ces composants. L'état de l'application du point de vue de chaque adaptation ne reflète que les changements explicitement effectués par cette dernière. Cette propriété est très importante lorsque plusieurs adaptateurs sont susceptibles de modifier l'application. Un mécanisme de synchronisation est utilisé pour isoler les modifications d'une adaptation des autres. Pour les systèmes de bases de données, un verrou sur une ligne ou une table est suffisant pour empêcher des accès concurrents à une même donnée. Par contre, lorsqu'une adaptation modifie un composant, la pose d'un verrou sur ce dernier empêche d'autres adaptations de modifier ce composant mais n'assure pas complètement la propriété de consistance. En effet, les propriétés du Quoi liées à la globalité d'une architecture (cycles et de points de non-déterminisme) peuvent toujours être mises en péril car deux adaptations visant deux composants différents peuvent être effectuées en parallèle (puisque pas de verrou posé)[2]. Pour assurer une mise en œuvre sûre des adaptations, les adaptations concurrentes qui s'appliquent à différents composants appartenant à un même assemblage sont donc mises en œuvre de façon séquentielle par la pose d'un verrou sur l'ensemble des composants de l'assemblage. Notons que ce verrou concerne uniquement les accès en mode « adaptation » et ne doit pas empêcher l'utilisation des composants de l'assemblage dans la mesure où cela respecte les propriétés du Quand.

Durabilité. Cette propriété est nécessaire pour garantir qu'une adaptation tolère les pannes (crash du disque, pannes de courant, etc). Le système sous-jacent est lui-même supposé durable (l'état des composants est persistant) et capable de mémoriser les adaptations afin de les réappliquer ultérieurement pour récupérer l'état (structure) des composants et leur connexions.

[2]Les propriétés de sûreté globales sont alors vérifiées sur un état ne prenant pas en compte d'autres adaptations en cours.

5.2 Modèle abstrait de sûreté d'adaptation

Cette section détaille brièvement le modèle abstrait de sûreté d'adaptation Satin et positionne cette modélisation par rapports à différents modèles tels que UML 2.0, les ADLs, les modèles à composants. Le modèle repose sur une abstraction de composants pour l'adaptation.

5.2.1 Présentation synthétique du modèle abstrait

Satin s'appuie sur la définition de **composant** du monde des ADLs[3] en précisant qu'il s'agit d'instances supportant au minimum l'envoi de messages. Un **port** permet d'expliciter les points de connexion entre composants[4]. Un port correspond généralement dans le monde des composants au niveau de granularité de l'interface. Dans le modèle Satin, la fonctionnalité est le niveau de granularité choisi pour exprimer les points de connexion. Ainsi, une fonctionnalité est désignée par le terme port dans la suite.

Un **template** (figure 5.1) définit une catégorie de composants aux mêmes caractéristiques. Une **implantation** est la représentation de la définition du composant exprimée dans une plate-forme spécifique (classes, descripteurs, ...). Un template peut être défini de différentes façons selon le contexte de création (au moment de l'adaptation ou au préalable à partir d'une ou plusieurs implantations et d'un ensemble d'interfaces éventuellement vide). Des précisions sur ce point sont données dans le chapitre 6.

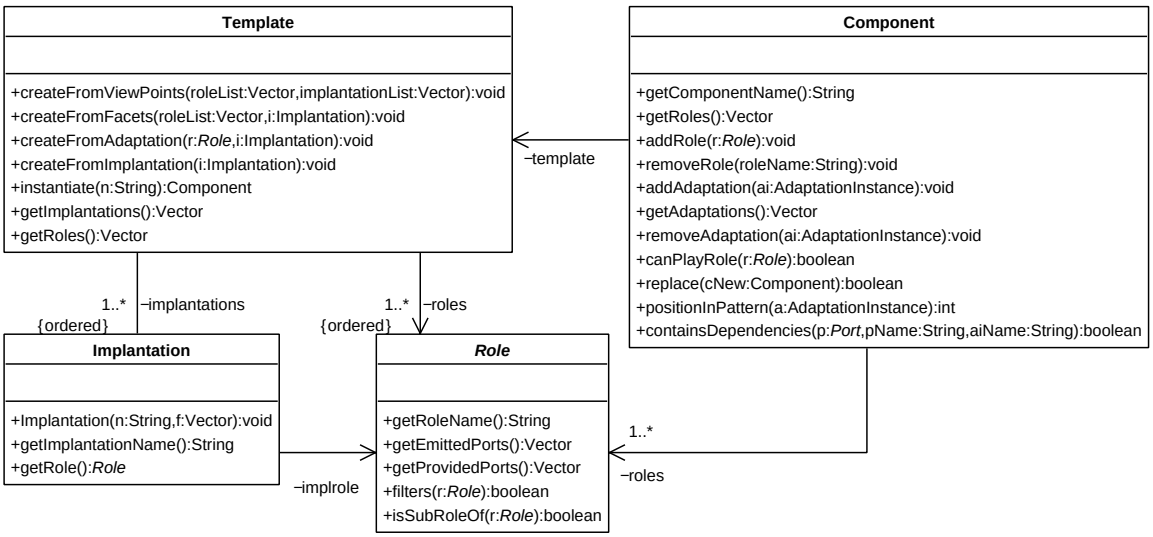

FIG. 5.1 – Representation UML de template, implantation et composant

L'aspect sûreté du modèle est ancré principalement sur deux concepts : 1) le **schéma d'adaptation**, une réification de la notion d'adaptation et 2) le **rôle**, une forme de typage dédiée à la prise en compte des adaptations. Pour ne pas alourdir ce chapitre, les notions de rôle et de schémas d'adaptation sont détaillées dans le chapitre 6 qui leur est consacré. Ces deux notions sont introduites ci-après de manière intuitive pour simplifier le discours dans la section suivante.

[3] "A component in an architecture is a unit of computation or a data store." [110]
[4] "Ports represent interaction points between a classifier and its environment." [91]

Concepts clé : schémas d'adaptation et rôles

Un schéma d'adaptation (figure 5.2) décrit des modifications de structure et de comportement de composants. Un schéma d'adaptation est donc un ensemble d'adaptations élémentaires définies sur un ensemble de rôles (les paramètres du schéma). Comme nous l'avons vu dans le chapitre 2, les types d'adaptations élémentaires sont de l'ordre de l'ajout et du retrait de port à un composant (qui entrent dans la famille d'adaptation **évolution des interfaces**), ajout, retrait ou remplacement de composants au sein d'un assemblage (qui entrent dans la famille d'adaptation **altération des assemblages**) et la modification du comportement associé à un port de composant (qui entre dans la famille d'adaptation **composition comportementale**), ...

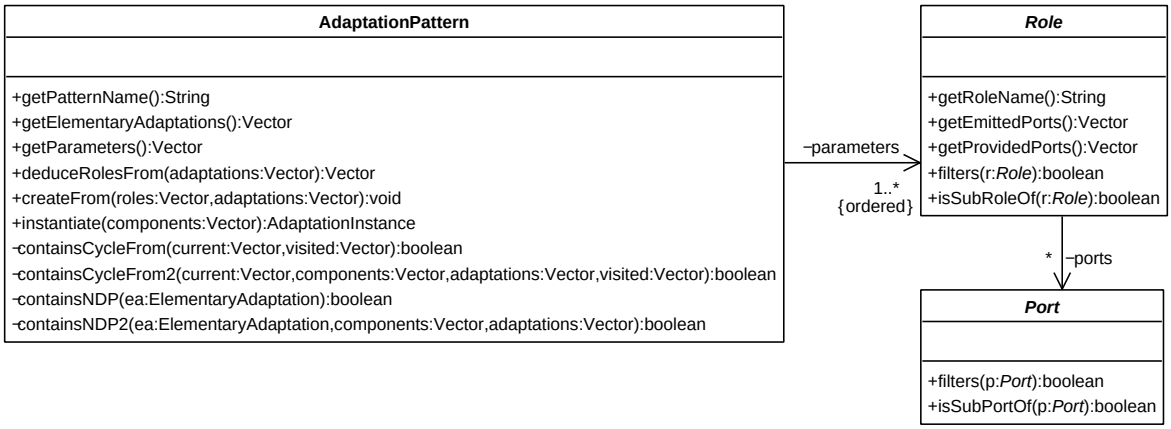

FIG. 5.2 – Représentation UML de schémas d'adaptation

Un **schéma d'adaptation** (figure 5.3, `AdaptationPattern`) représente l'unité d'application et de réutilisation des adaptations élémentaires. Appliquer un schéma d'adaptation à un ensemble de composants a pour effet de modifier la structure et le comportement de ces derniers. Lorsqu'un schéma est applicable (les propriétés de sûreté à vérifier sont préservées), une instance d'adaptation est créée. Une **instance d'adaptation** (figure 5.3, `AdaptationInstance`) désigne donc une application particulière d'un schéma à un ensemble donné de composants et designe un ensemble d'assemblages de composants.

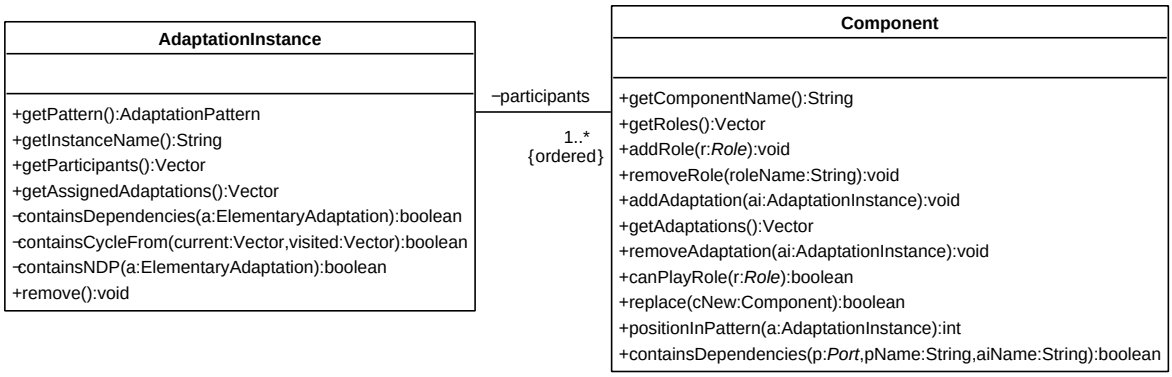

FIG. 5.3 – Représentation UML d'instance d'adaptation

Appliquer un schéma d'adaptation peut être une action réversible. Les besoins des utilisateurs d'une application pouvant évoluer et certaines adaptations pouvant ne plus être souhaitées, il est possible de défaire les modifications exercées sur les composants. Plutôt que de supprimer les adaptations passées en appliquant de nouvelles adaptations pour annuler les effets de précédentes adaptations, il suffit de « désappliquer » les schémas d'adaptation c'est à dire l'opération inverse à l'application d'un schéma. Ainsi, adapter un ensemble de composants consiste à appliquer ou désappliquer un ou plusieurs schémas d'adaptation à ces composants.

Pour résumer, l'application d'une adaptation élémentaire de type **ajout de port** à un composant *c* revient à l'ajout du port fourni au rôle de *c* et la désapplication de cette adaptation à *c* revient au retrait de ce port fourni du rôle de *c* et correspond à une adaptation élémentaire de type **retrait de port**. Pour une adaptation élémentaire de type **contrôle de port** permettant de modifier le comportement associé à un port, l'application du schéma sur un composant *c* revient à composer le nouveau contrôle avec le contrôle existant sur le port du composant et la désapplication du schéma à *c* revient à recomposer l'ensemble des contrôles exercés sur le port de *c* à l'exception du contrôle défini dans le schéma à désappliquer.

Un **rôle associé à un schéma d'adaptation** indique les ports qu'un composant doit fournir ou requérir pour jouer un rôle donné dans une adaptation de la même façon qu'un acteur doit avoir un certain nombre de caractéristiques exigées par le metteur en scène pour pouvoir jouer un certain rôle dans une pièce de théâtre.

Un **rôle associé à un composant** à un instant t décrit les fonctionnalités offertes par le composant, les fonctionnalités requises par le composant dans le cadre de ses adaptations ainsi que les différentes adaptations qui lui ont été appliquées au travers des schémas d'adaptation.

Etant instancié à partir d'un template, un composant peut être associé à un ou plusieurs rôles (suivant la manière dont a été créé le template). Tous les composants issus d'un même template ont initialement des rôles équivalents. Ensuite, les rôles de chaque composant évoluent, par adaptation, indépendamment des rôles des autres composants issus du même template.

Un modèle abstrait décoré pour préserver les propriétés de sûreté

Les propriétés de sûreté peuvent être exprimées directement dans le modèle abstrait sous forme de contraintes OCL. La description complète est faite en section 7. Intuitivement, l'intérêt d'avoir un modèle abstrait incluant les propriétés est de :

– les exploiter pour chaque plate-forme sans les réécrire de manière spécifique,
– les prouver une seule fois au niveau du modèle abstrait.

5.2.2 Satin versus UML, ADLs et modèles à composants

La notion de rôle est à rattacher à la notion d'interface de collaboration de même nom définie dans UML 2.0 [91] et dans le paradigme OORASS (Object Oriented Role Analysis, Synthesis and Structuring) [9] à l'exception près que les collaborations, dans notre cas, sont les assemblages liant les composants par adaptation. Les rôles peuvent aussi être vus comme des facettes au sens des modèles RM-ODP [55, 54, 53, 52] et CCM [89]. Dans les ADLs telles que UniCon [124] ou ACME [43], un rôle représente l'extrémité d'un connecteur. Or, un connecteur représente la « glue » entre deux composants dans un contexte donné et peut être rapproché au concept de collaboration. Nos rôles englobent donc aussi cette définition.

Les schémas d'adaptation permettent une représentation abstraite des adaptations élémentaires. D'après la classification des connecteurs proposée par Medvidovic [77], les adaptations élémentaires d'un schéma d'adaptation peuvent être assimilées à des connecteurs implicites dont les extrémités (les rôles du schéma) sont attachés à des interfaces (les rôles des composants auxquels on applique le schéma).

Dans notre cas, la notion de port se rapproche de la notion de fonctionnalité et ne désigne pas une interface comme dans la plupart des modèles à composants et certains ADLs. De plus, nous ne modélisons pas les interfaces requises des composants car dans notre modèle, les composants représentent toute entité logicielle (objet, composant, acteur, agent, ...). Cependant, les les fonctionnalités utilisées dans le contexte d'une adaptation peuvent être vues comme un sous-ensemble des interfaces requises.

5.3 Exemple d'utilisation du modèle abstrait Satin

Pour mieux comprendre le modèle abstrait Satin, prenons une application d'agendas.

5.3.1 Composants à adapter

Dans la suite, nous considérons deux composants représentant respectivement l'agenda de Anne Marie et l'agenda de l'équipe : *agendaAM* et *agendaEquipe*. Le composant agenda de l'équipe a pour rôle `AgendaDeBase` qui fournit les ports de manipulation de rendez-vous (`addRdv`, `removeRdv` et `getRdv`). L'agenda de Anne-Marie a pour rôle `AgendaEvolué` qui permet en plus de gérer les collisions de rendez-vous (`addRdv`, `removeRdv`, `getRdv`, `isFree` et `printError`) (cf. figure 5.4).

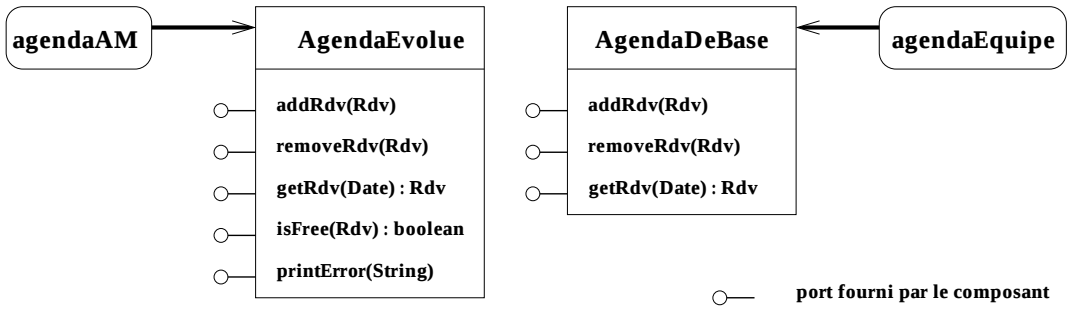

FIG. 5.4 – *agendaAM* et *agendaEquipe*

Dans cet exemple, les composants sont supposés créés au préalable (et non pas au moment d'une première adaptation de composants). La création de composants entraîne la vérification des propriétés $\mathbf{P_{1a}}$, $\mathbf{P_2}$ et $\mathbf{P_{4a}}$ d'après le tableau 5.1. En particulier, nous devons vérifier que chaque composant à créer correspond à un composant concret qui implémente les fonctionnalités correspondant aux ports décrits dans son rôle. Pour cela, les templates permettant de créer ces composants doivent respecter la propriété $\mathbf{P_{1a}}$ sur l'adéquation des rôles par rapport à l'implantation. On suppose disposer de deux implantations respectivement `AgendaDeBaseImpl` et `AgendaEvolueImpl` fournissant un code d'implémentation pour chaque port des rôles des deux composants *agendaEquipe* et *agendaAM*. Les templates doivent respecter la propriété $\mathbf{P_2}$ sur la garantie de visibilité. Cette propriété est assurée car les deux composants n'ont qu'un rôle donc tous les ports qu'ils proposent sont accédés de la même façon. La propriété $\mathbf{P_{4a}}$ sur l'uniformité d'application des adaptations est assurée pour la même raison. La création de ces composants ne viole donc aucune propriété.

5.3.2 Adaptation pour la synchronisation d'Agendas

Supposons que Anne-Marie veuille synchroniser son agenda avec celui de l'équipe. C'est à dire qu'elle veut ajouter les rendez-vous de l'équipe dans son propre agenda lorsque la place est disponible. Evaluer si cette adaptation est possible revient à vérifier l'applicabilité du schéma

d'adaptation Synchronisation aux composants *agendaAM* et *agendaEquipe*. Le schéma d'adaptation Synchronisation est décrit ci-dessous à l'aide du langage ASL (cf chapitre 6). Il comporte une adaptation élémentaire de type **modification comportementale**. Cette adaptation élémentaire stipule que tout ajout de rendez-vous à un agenda sur lequel on se synchronise entraîne un ajout de rendez-vous sur l'agenda à synchroniser seulement si le créneau du rendez-vous est libre sinon il signale un problème de conflit de rendez-vous.

Le schéma s'applique à deux paramètres. Le premier paramètre doit être conforme au rôle AgendaSynchronisant et le second au rôle AgendaSynchronisé. Ces rôles sont déduits de l'adaptation élémentaire décrite dans le schéma. En l'occurrence AgendaSynchronisant fournit le port addRdv et AgendaSynchronisé fournit les ports addRdv, isFree et printError. Ainsi, tout composant comportant au moins un port conforme (au sens du filtrage) à addRdv (resp. addRdv, isFree et printError) peut jouer le rôle AgendaSynchronisant (resp. AgendaSynchronisé). La notion de **conformité au sens du filtrage** est explicitée dans le chapitre 6.

```
adaptationPattern Synchronisation(AgendaSynchronisant a1, AgendaSynchronisé a2) {
 modifyPort a1.addRdv(Meeting m) -> if (a2.isFree(m)) then
                                       a1._call(m); a2.addRdv(m)
                                    else
                                       a1._call(m); a2.printError("non synchronisé")
                                    endif
}
```

Le schéma d'adaptation Synchronisation est applicable seulement si les propriétés de sûreté $\mathbf{P_2}$, $\mathbf{P_3}$, $\mathbf{P_{4a}}$, $\mathbf{P_{4b}}$, $\mathbf{P_{4c}}$ et $\mathbf{P_5}$ sont préservées (cf. le tableau 5.1). Par contre, les propriétés $\mathbf{P_0}$, $\mathbf{P_{1a}}$, $\mathbf{P_{1b}}$ et $\mathbf{P_6}$ n'ont pas à être vérifiées car elles ne concernent pas le type d'adaptation élémentaire visé par le schéma Synchronisation qui est de la famille **modification comportementale**.

5.3.3 Application de l'adaptation de synchronisation à deux agendas donnés

Dans l'application du schéma, *agendaAM* joue le rôle de l'agenda synchronisé et *agendaEquipe* celui de l'agenda synchronisant. Les propriétés de sûreté $\mathbf{P_2}$ à $\mathbf{P_5}$ sont préservées par cette adaptation. La propriété $\mathbf{P_2}$ sur la garantie de visibilité est assurée car les ports de chaque composant utilisé dans l'adaptation élémentaire appartiennent au même rôle du composant en question.

La propriété $\mathbf{P_3}$ sur la consistance des assemblages est préservée puisque l'adaptation élémentaire définie par le schéma d'adaptation crée un assemblage où les composants sont conformes aux rôles qu'ils doivent jouer. En effet, le rôle de *agendaAM* est conforme au rôle AgendaSynchronisé car à tous ports fournis de AgendaSynchronisé correspond un port fourni du rôle de *agendaAM*[5]. De la même façon, le rôle de *agendaEquipe* est conforme au rôle AgendaSynchronisant.

La propriété $\mathbf{P_{4a}}$ sur l'uniformité d'application des adaptations ne peut être mise en défaut dans cet exemple. Les composants considérés n'ont qu'un seul rôle, le schéma va donc bien s'appliquer à tous les ports concernés du composant.

La propriété $\mathbf{P_{4b}}$ sur la compatibilité des adaptations élémentaires est préservée puisqu'il n'existe pas d'autres adaptations élémentaires du même type déjà appliquées sur le port addRdv de *agendaEquipe* (la question de la compatibilité ne se pose donc pas). La propriété $\mathbf{P_{4c}}$ sur les points de non-déterminisme n'a pas besoin d'être vérifiée dans la mesure où les ports des composants peuvent être appelés dans n'importe quel ordre.

Enfin, la propriété $\mathbf{P_5}$ sur les cycles est préservée puisque les ports isFree, printError et addRdv de *agendaAM* n'ont jamais été adaptés (ils ne sont donc pas supposés utiliser, de manière indirecte ou non, le port addRdv de *agendaEquipe*).

[5] En supposant également que Rdv soit conforme à Meeting.

Le schéma `Synchronisation` étant applicable, l'adaptation est donc possible. Un assemblage est créé (*agendaAM* et *agendaEquipe* sont connectés) et le rôle de *agendaEquipe* est modifié pour mémoriser cette adaptation.

Notons que si nous avions permuté les rôles des composants, l'adaptation n'aurait pas été autorisée : la propriété $\mathbf{P_3}$ sur la consistance des assemblages n'est pas préservée par cette adaptation car *agendaAM* ne peut pas jouer le rôle `AgendaSynchronisant`. En effet, le rôle de *agendaAM* ne fournit pas les ports `printError` et `isFree` et n'est donc pas conforme au rôle `AgendaSynchronisant`.

Enfin, l'application du schéma d'adaptation a pour effet de créer une instance d'adaptation représentant un ensemble d'assemblages des composants adaptés. La figure 5.5 permet de visualiser graphiquement l'instance d'adaptation ainsi créée. A chaque fois que nous avons besoin de désigner une instance d'adaptation, nous utilisons cette représentation.

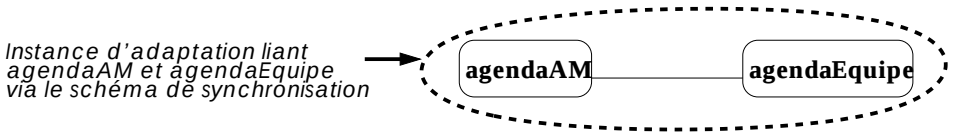

FIG. 5.5 – Instance d'adaptation liant *agendaAM* et *agendaEquipe*

Cet exemple est complété dans le chapitre suivant qui détaille le contenu des rôles et explique quelles sont les modifications de rôles que l'application d'un schéma implique.

5.4 Positionnement de Satin dans le cadre de l'IDM

Comme nous l'avons vu dans le chapitre 2, la manière dont sont exprimées les adaptations varie selon les outils. Celles-ci peuvent être programmées en utilisant diverses APIs comme dans le cas des modèles à composants CORBA [89] et Fractal [86] ou bien décrites à l'aide d'un langage spécifique comme ISL [12] ou le langage de spécification des filtres de composition [13]. De plus, nous avons constaté que chaque plate-forme ne prend en charge qu'une partie des types d'adaptations possibles. D'autre part, nous constatons qu'en fonction des langages et des plates-formes cibles, les notions de conformité vis-à-vis d'une « interface » (au sens du type) varient [14, 37]. Le caractère dynamique des applications actuelles renforce cette hétérogénéité dans la vérification de la conformité puisque les composants doivent pouvoir dynamiquement être assemblés en respectant des « types » qui eux-mêmes se créent tandis que l'application évolue (découverte dynamique de services, chargement de classe dynamique, ...).

Un de nos objectifs étant de concevoir un modèle suffisamment abstrait pour être indépendant des plates-formes mais aussi suffisamment concret pour être opérationnel, nous avons souhaité clairement séparer les éléments de modélisation propres à notre problématique des éléments que nous utilisons. Ainsi, le modèle abstrait vu en 5.2 regroupe les différents éléments de modélisation nous permettant de répondre à la problématique de la sûreté des adaptations. Les éléments de ce modèle sont décorés de contraintes préservant les propriétés de sûreté définies en 5.1. Cependant, comme il est indispensable de monter d'un niveau d'abstraction pour modéliser une solution générale, il est également incontournable de redescendre d'un niveau pour que cette solution soit utilisable. Dans le cas contraire, nous ne ferions que proposer un nième modèle isolé des autres.

L'IDM regroupe un ensemble d'approches du processus de développement logiciel centrées sur des modèles telles que la programmation générative [32], les langages spécifiques domaines (DSL) [118], le MIC (Model Integrated Computing), les usines à logiciels (Software Factories) [47] et la MDA [90]. Dans la suite, nous nous appuyons sur la démarche MDA pour expliciter comment prendre en compte un ensemble de concepts liés aux plates-formes à composants adaptables et nécessaires à la concrétisation du modèle abstrait.

La section 5.4.1 introduit les principaux concepts sur lesquels repose la MDA et les outils que nous avons utilisés pour la mettre en œuvre. Puis, la section 5.5 décrit comment nous avons raffiné le modèle abstrait pour prendre en compte la spécificité des plates-formes cibles essentiellement au niveau des adaptations élémentaires autorisées et du typage des composants. Enfin, nous passons en revue quelques approches du domaine des composants et des aspects autour de l'adaptabilité qui suivent la démarche MDA dans la section 5.4.2 pour évaluer la pertinence de notre modèle abstrait.

5.4.1 Concepts de base de la MDA et outils

La Model Driven Architecture (MDA, architecture dirigée par les modèles) entre dans le cadre de l'IDM et vient remplacer l'ancienne organisation OMA (Object Management Architecture) préconisée par l'OMG. La MDA vise à construire des modèles indépendants des plates-formes techniques pour préparer l'évolutivité des systèmes, permettre une meilleure capitalisation des applications et soustraire les concepteurs de la dépendance aux technologies d'exécution. Indépendantes des choix de plates-formes, les applications basées sur la MDA peuvent alors supporter l'intégration, l'interopérabilité et l'évolution au fur et à mesure de l'évolution des besoins, et de l'apparition et la disparition des solutions technologiques.

Afin d'organiser les modèles définissant une application ou un système, la MDA propose un découpage selon deux préoccupations majeures : 1) l'expression des fonctionnalités d'une application métier ; 2) l'expression des spécificités technologiques d'une mise en œuvre de ces fonctionnalités. Ainsi, un PDM (Platform Description Model) est un modèle décrivant une technologie particulière. Un PIM (Platform Independent Model) se concentre sur la modélisation des éléments métiers destinés à résoudre un problème donné d'un domaine spécifique (finance, médical, e-commerce, ...) en faisant abstraction des détails techniques. A l'inverse, un PSM (Platform Specific Model) introduit les détails spécifiques à une technologie c'est à dire à un PDM. Un PSM est la manière de représenter une solution à un problème dans un environnement donné.

Le passage d'un PIM à un PSM se fait successivement en décrivant des règles de transformation de modèles. Cependant, les transformations ne sont pas uniquement nécessaires pour passer d'un PIM vers un PSM, mais aussi pour traduire un modèle issu d'un métamodèle vers un autre métamodèle ou bien pour modifier un modèle en restant dans le langage induit par son métamodèle. Ainsi il existe deux types de transformation : le raffinement d'un modèle *A* en un modèle *B* et la projection d'un modèle *A* vers un modèle *B*. La principale différence entre ces deux types de transformation est que dans le cas du raffinement, on reste dans le même métamodèle[6] contrairement à la projection (figure 5.6).

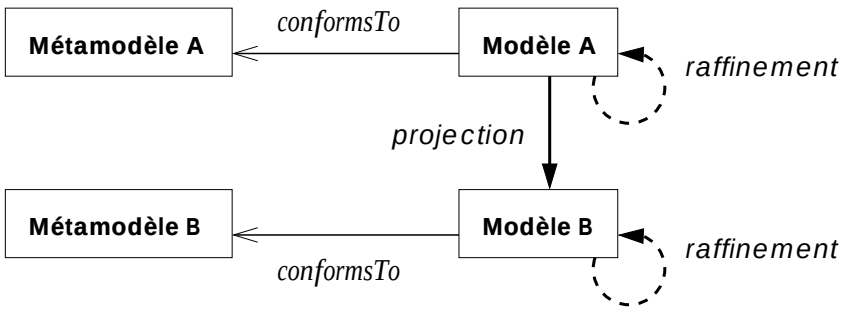

FIG. 5.6 – Processus de transformation de la MDA

[6]Le métamodèle est au modèle ce que la légende est à la carte géographique : il permet de l'interpréter [23].

Pour expliciter le processus de raffinement, nous nous basons sur la méthodologie Picore [71]. Picore est un environnement de modélisation permettant la définition de modèles par raffinement[7]. Un modèle est un ensemble de classes composées d'attributs et d'opérations. Le *modèle de base* est enrichi soit directement par raffinement soit par intégration avec un autre modèle. On obtient un *modèle raffiné* qui peut ensuite être considéré comme le nouveau modèle de base et être raffiné à son tour. Deux relations permettent de définir un raffinement : *refines* et *using*.

La relation *refines* entre deux modèles est une forme de relation d'héritage pour les modèles car tous les éléments du modèle de base appartiennent au modèle raffiné, qui peut aussi en posséder d'autres. De même, la relation *refines* entre classes met en correspondance les classes du modèle de base et celles du modèle raffiné qui correspondent au même élément (i.e. : ayant le même nom dans les deux modèles). Tous les attributs de la classe de base appartiennent à la classe raffinée, qui peut aussi posséder d'autres attributs. Notons que l'on peut aussi raffiner un modèle en ajoutant de nouvelles classes, ajouter de l'héritage, etc. La relation *using* permet d'utiliser les concepts d'un autre modèle. Plutôt que de définir au niveau du modèle de base les concepts utiles au raffinement, ces derniers sont définis dans un *modèle d'extension*. Ces concepts sont alors utilisables dans le modèle raffiné par intégration du modèle de base et du modèle d'extension.

Picore étant utilisé pour décrire les spécificités de raffinement du modèle abstrait Satin, des exemples syntaxiques sont donnés dans la suite.

5.4.2 Quelques travaux autour de la MDA et de l'adaptabilité

Dans le domaine des composants et des aspects, plusieurs travaux utilisent la MDA. Les approches ci-après s'attachent à modéliser une partie de l'univers dans un but particulier. Ainsi, Barais et al [10] s'intéressent à l'étape de construction et de modélisation des applications à base de composants, Hachani et al [48] abstraient la notion de séparations des préoccupations et les concepts sous-jacents pour proposer un cadre permettant la conception d'aspects, Flissi et al [40] modélisent la phase de déploiement des composants et Nano et al [83] se concentrent sur l'intégration de services.

Méta modélisation des langages d'aspects. Cette approche propose d'utiliser les mécanismes et concepts des approches orientées aspects pour améliorer l'implémentation des patrons de conception du GoF [48]. La proposition s'appuie sur une méta modélisation des langages d'aspects afin d'exprimer ces nouvelles structures de patrons indépendamment d'un langage spécifique. Ce cadre de conception étend celui d'UML dans le but d'améliorer la réutilisation de patrons dès la phase de conception. Des règles de transformation permettent de passer du métamodèle général à des PSM spécifiques à chaque langage d'aspects tels que AspectJ [59] ou HyperJ [115] et donc de générer une modélisation « technologique » d'un patron particulier à partir de sa modélisation « pure ». Bien que motivé par le besoin de trouver un moyen d'expression des structures de patrons par aspects indépendamment des langages à aspects, ce travail peut s'appliquer à n'importe quelle modélisation par aspects.

SafArchie. Ce projet vise à permettre une spécification assistée de l'architecture d'une application. SafArchie [10] est basé sur une méta modélisation des architectures logicielles permettant d'analyser l'architecture d'un point de vue structurel et comportemental. Le modèle d'architectures se découpe lui-même en deux modèles nommés respectivement modèle de type de composants et modèle logique permettant de séparer la spécification des invariants du système et sa dynamique. SafArchie permet ainsi d'analyser un assemblage d'un point de vue structurel et comportemental et la construction d'architectures logicielles typées indépendamment des ADLs et des plates-formes à composants sous-jacentes. Une description de l'architecture écrite

[7] La MDA préconise d'utiliser le MOF [88] pour définir les modèles. Cependant, les descriptions de modèles Picore sont beaucoup plus simples à écrire et comprendre. De plus, chaque concept Picore a un équivalent dans le MOF et l'outil propose un traducteur pour générer les descriptions en MOF.

avec SafArchie peut-être projetée vers des plates-formes d'exécution possédant les concepts de composant, composite et connecteur en vue de garantir la traçabilité des concepts architecturaux au niveau de l'implantation. Les plates-formes disponibles sont pour le moment ArchJava [5] et Julia [21], l'implantation de référence de Fractal [86].

Méta modélisation du déploiement. Cette approche consiste à générer des machines de déploiement pour les plates-formes à composants. L'activité de déploiement peut alors être vue comme un ensemble de taches (installation, instanciation, configuration) à exécuter dans un ordre défini et dont les principales données sont les fabriques à composants, leurs instances et leurs interfaces. Le modèle abstrait de déploiement (PIM) s'appuie alors sur un profil UML de workflow définissant les stéréotypes de classe *tache* et *donnée* et les stéréotypes d'association *entrée*, *sortie* et *dépendance*. Des raffinements de ce modèle permettent d'obtenir des PIMs de déploiement spécialisés pour un modèle de composants donné : ils enrichissent le PIM avec des informations métier spécialisées pour un modèle de composants. Ensuite, la transformation d'un PIM raffiné permet d'obtenir le modèle déploiement spécifique à une technologie de mise en œuvre (PSM) qui est projeté vers la plate-forme d'exécution. Par exemple, le PIM de déploiement spécifique au CCM transformé en PSM spécifique à Fractal conduit à la génération d'une machine de déploiement pour des applications s'exécutant sur une plate-forme CCM avec comme technologie sous-jacente, une plate-forme Fractal.

MicM. Le projet MicM [83] vise à aider à la définition et à l'intégration de services sur multiples plates-formes. Il est basé sur un modèle de composants abstraits qui définit des opérations pour l'évolution structurelle et comportementale des composants. Ce modèle permet, d'une part, la définition séparée des intégrations de services et, d'autre part, la composition des intégrations de services en fonction de l'application. MicM introduit un mécanisme de composition permettant de détecter des incompatibilités de composition entre différentes intégrations de services. La projection dans les plates-formes à composants se définit à l'aide d'un modèle intermédiaire de composants concrets qui permet de combiner les évolutions des intégrations de services et des informations issues de la plate-forme à composants cible. La projection permet de détecter des incompatibilités entre les intégrations de services et la plate-forme cible.

Positionnement de Satin par rapport à ces modèles. « A model represents reality for the given purpose » [105]. Afin de montrer l'impact du but recherché dans le travail de modélisation, nous comparons les modèles que nous venons d'étudier et Satin. Ainsi, nous montrons que bien que ces travaux soient effectués dans des domaines très proches (composants, aspects, et adaptations), les modèles mettent rarement la même chose derrière les mêmes concepts car ils n'ont pas les mêmes objectifs. Le tableau 5.2 récapitule les principales similitudes et différences entre ces cinq modèles.

	Modélisation des concepts			
	Type	**Instance**	**Implantation**	**Autres**
Modèle d'aspects	Fonctionnalités offertes	Non	Classifier (classe d'implantation)	Crosscutting concern, crosscutting feature
SafArchie	Fonctionnalités offertes, fonctionnalités requises, contrats comportementaux	Oui		Connecteurs, composites
MicM	Fonctionnalités offertes, visibilité	Non	Classe d'implantation, proxy, squelette, intercepteur	Intégrateurs, règles structurelles et comportementale
Modèle de déploiement	Fonctionnalités offertes, fonctionnalités requises	Oui	Classe d'implantation	Tache de déploiement
Satin	Fonctionnalités offertes, contrats d'adaptation	Oui	Classe d'implantation	Schéma d'adaptation, adaptation élémentaire

TAB. 5.2 – Comparaison du modèle Satin avec les quatre modèles présentés précédement

Tous les modèles ont en commun l'utilisation de la notion de fonctionnalités offertes (interface au sens classique du terme). Par contre, seuls SafArchie [10] et le modèle de déploiement [40] modélisent les fonctionnalités requises des composants afin de décrire les assemblages de composants. Le modèle Satin n'utilise pas cette notion car le but n'est pas de fiabiliser les assemblages préexistants mais de contrôler que des assemblages créés par adaptations soient consistants (la notion de ports émis est utilisé à cet effet). D'autre part, SafArchie introduit son propre langage afin de décrire les contrats comportementaux des composants. A l'inverse, le modèle Satin s'appuie sur une abstraction de la notion de conformité pour conserver l'indépendance vis-à-vis des plates-formes. Enfin, le modèle Satin ne modélise que les adaptations qui modifient l'interface « accessible » du composant et ne gère pas la mise en œuvre des adaptations dans les composants eux-mêmes mais seulement l'applicabilité de celles-ci. Ainsi, alors qu'un composant du point de vue de MicM [83] peut présenter différentes visibilités (par exemple une interface distante, privée ou de service), la notion de visibilité n'apparaît pas au niveau du modèle Satin.

L'implantation MicM peut correspondre à une classe d'implantation ou à un ensemble d'objets d'implémentations tels que interface, squelette, proxy et intercepteur. Des éléments tels que les squelettes ou les proxies sont modélisés car, pour pouvoir valider l'intégration de services, il est important de modéliser le « lieu » de génération de cette intégration au niveau des plates-formes. Alors que seule la classe d'implantation d'un composant est utile pour les autres modèles.

Satin modélisant l'étape d'exécution du cycle de vie des application, il est donc logique qu'il prenne en compte les instances de composants. Ce choix se justifie aussi par le fait que représenter les instances permet d'exprimer les adaptations et de les appliquer à différentes granularités. A l'inverse, dans le modèle MicM, les instances de composants ne sont pas représentées parce que l'intégration de service et le déploiement sont pris en charge un niveau d'un type de composant (indifféremment pour toutes les instances). Le modèle de déploiement qui se situe pourtant au même moment du cycle de vie des applications que MicM utilise le concept d'instances puisqu'il prend en compte l'instanciation, la configuration et d'activation des instances de composants (ce qui justifie que ce concept soit présent). Quant à la modélisation des préoccupations transversales (ou modèle d'aspects) proposée par [48], elle ne réifie pas le concept d'instance car ce modèle se positionne à l'étape de conception des applications. A nouveau, bien que se positionnant également au niveau conception, le modèle de type d'architecture de SafArchie met en évidence un certain nombre de contraintes et le modèle d'instances de composant de SafArchie représente une vision statique de l'architecture logicielle telle qu'elle est déployée et exécutée à un instant t et qui doit vérifier l'ensemble des contraintes.

Un des points communs entre Satin et le modèle d'aspects est l'abstraction de la notion d'adaptation. Satin s'appuie sur cette abstraction pour établir des propriétés de sûreté et le modèle d'aspects pour modéliser les patrons de conception par aspects. Aussi les notions de *crosscutting concern* et de schéma d'adaptation sont-elles très proches. Il en est de même pour les notions de *crosscutting feature* et d'adaptation élémentaire. Mais contrairement au schéma d'adaptation, *crosscutting concern* et *classifier* font partie de la même hiérarchie de classes ce qui fait qu'un aspect lui-même peut être modifié par un autre aspect comme s'il était un objet.

5.5 Positionnement du modèle abstrait Satin vis-à-vis du processus de raffinement

Cette section décrit comment nous pouvons raffiner le modèle abstrait Satin pour prendre en compte la spécificité des plates-formes cibles essentiellement au niveau des adaptations élémentaires autorisées et du typage des composants.

5.5.1 Modèles d'extension et exemples de concrétisation

Cette section décrit quels modèles d'extensions (préoccupations à intégrer au modèle abstrait) permettent de rapprocher le modèle abstrait Satin des plates-formes concrètes. Au vue de l'exemple décrit en 5.3, nous identifions essentiellement deux points relatifs à la plate-forme cible : la conformité des composants à un rôle et l'expression des adaptations élémentaires.

- Il est important de résoudre les contraintes d'interdépendances entre les rôles de différents composants auxquels on souhaite appliquer un schéma d'adaptation avant d'accepter l'application de ce dernier par exemple. Cette vérification est en particulier modélisée dans la classe UML `Role` par l'opération `isSubRoleOf`. La notion de conformité nécessaire ici (au sens de la substituabilité des comportements) fait référence à différentes formes de contrats. Ainsi, si l'on reprend la classification de Beugnard et Al. [15], la notion de conformité au sens du contrat syntaxique va se décliner différemment suivant les plates-formes : absence de sous-typage, conformité au sens de l'héritage ou polymorphisme d'inclusion. De la même manière, la notion de conformité au sens du contrat comportemental va se baser sur des préconditions et postconditions ou tout autre formalisme permettant de décrire le comportement attendu des composants. Le même principe s'applique aux contrats de synchronisation et aux contrats de qualité de service. Or, les plates-formes ne prennent pas en charge tous les niveaux de contrat et ne se basent pas sur les mêmes modèles pour gérer la mise en œuvre de la notion de conformité. Il se dégage donc naturellement le besoin d'un modèle d'extension qui soit **un modèle de contrats (**`ContractExtension`**)**.

- Toute adaptation se définit comme un ensemble adaptations élémentaires (modélisées par l'opération `getElementaryAdaptations` dans la classe UML `AdaptationPattern`). Il faut donc pouvoir vérifier que l'ensemble des propriétés des composants utilisées dans une adaptation élémentaire sont présentes au niveau de ces composants. Par ailleurs, un composant peut subir plusieurs adaptations élémentaires successives. Il est donc important de pouvoir déterminer si la composition d'un ensemble d'adaptations élémentaires est possible et d'en calculer le résultat dans ce cas. Il est également nécessaire de savoir lorsqu'une adaptation introduit de l'indéterminisme. Enfin, les adaptations élémentaires sont exercées sur les composants via des coupes. En effet, les coupes servent à exprimer l'ensemble des composants qui sont concernés par une adaptation. Or, la manière dont les adaptations élémentaires et les coupes sont exprimées et mises en œuvre varient suivant les plates-formes. De même, les types d'adaptations élémentaires pris en charge diffèrent d'une plate-forme à une autre : ajout ou retrait de méthode, ajout, retrait ou remplacement de composants au sein d'un assemblage, modification comportementale d'une fonctionnalité sont autorisés ou interdits selon les cas. Il se dégage donc naturellement le besoin d'un modèle d'extension qui soit **un modèle d'adaptations élémentaires et de coupes (**`AdaptationExtension`**)**.

Le tableau 5.3 regroupe quelques exemples de concrétisation de modèles d'extension pour certaines plates-formes que nous avons étudiées en 2. Lorsque le modèle de coupes correspond à une signature de méthode, cela signifie que l'application d'une adaptation se fait directement en spécifiant le composant et la méthode concernée. Lorsque le modèle de coupe se base sur des expressions régulières, les composants et / ou méthodes peuvent être choisis de façon générique. Le modèle d'adaptation élémentaire correspond soit à une API, soit au modèle associé à un langage de description des adaptations élémentaires. Ainsi, Jac autorise l'ajout et la modification comportementale de fonctionnalité tandis que Noah autorise seulement la modification comportementale. Fractal et CCM autorisent l'ajout, le retrait de composant ou toute combinaison des deux types d'adaptations élémentaires alors que Sofa n'autorise que le remplacement de composant.

	Modèle de coupes et d'adaptations élémentaires		Modèle de contrats
CCM/ OpenCCM	Signature de méthodes	API d'adaptation : (connect, disconnect)	Conformité au sens du typage nominal et prise en compte de la nature des ports (synchrone ou asynchrone)
Fractal/ Julia	Signature de méthodes	API d'adaptation : (bindFc, unBindFc, addFcSubComponent, removeFcSubComponent)	Polymorphisme d'inclusion sur les interfaces et prise en compte de la nature des ports (serveur ou client)
Sofa/ DCUP	Signature de méthodes	Modèle DCUP	Contrat de synchronisation basé sur les protocoles Sofa
Noah/ EJB	Expressions régulières sur les noms de méthodes	Les types d'opérateurs de ISL : (sequence, condition, delegate, ...)	Conformité au sens de l'héritage d'interfaces et de classes Java
CF/ Compose*	Expressions régulières sur les noms de classes et de méthodes	Les types de filtres : (Meta, Dispatch, Error, ...)	Dépend du langage dans lequel est écrite l'application

TAB. 5.3 – Exemples de concrétisation des modèles d'extension

5.5.2 Processus de raffinement du modèle abstrait : approche classique

Il n'est pas possible de représenter les différents types de contrats au niveau du modèle abstrait : cela oblige inévitablement à choisir une concrétisation particulière du modèle de contrat et de la notion de conformité associée. De même, une modélisation des adaptations élémentaires au niveau du modèle abstrait fige les types d'adaptations élémentaires supportés. Nous n'avons pas fait ces choix et, en ce sens, le modèle abstrait est bien indépendant plates-formes. Cependant, pour devenir opérationnel, ce modèle doit pouvoir se rapprocher des plates-formes visées.

Ainsi, pour prendre en compte les spécificités des plates-formes, la solution « classique » consiste à raffiner le modèle abstrait qui est alors considéré comme le modèle de base avec des concrétisation des modèles d'extension définis précédemment. Ainsi, la diversité, à la fois dans la manière d'exprimer les adaptations ou les contrats, peut-elle être prise en compte en raffinant le modèle de base pour chaque concrétisation des modèles d'extension. Dans ce cas, nous obtenons *n* versions raffinées du modèle abstrait. La figure 5.7 décrit comment le modèle abstrait peut être raffiné pour une plate-forme.

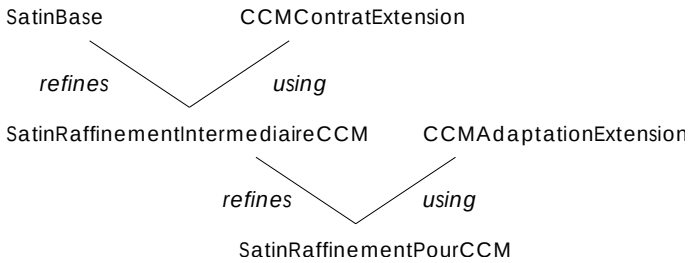

FIG. 5.7 – Processus de raffinement de Satin pour le modèle CCM

Notre conviction est que nous n'avons pas besoin d'exprimer tous les raffinements possibles. En effet, ce qui important au niveau du modèle raffiné ne concerne pas les éléments même du modèle d'extension mais les interactions entre le modèle d'extension et celui qui l'utilise. Il suffit de monter d'un niveau d'abstraction pour réaliser que seules les opérations attendues sur ces modèles d'extension ont de l'importance : les concrétisations changent mais l'« interface » permettant d'utiliser le modèle d'extension reste la même. Par exemple, en ce qui concerne le modèle d'extension pour les contrats, seule l'opération réalisant la vérification de la conformité entre deux contrats a de l'importance indépendamment de la concrétisation des contrats.

5.5.3 Processus de raffinement du modèle abstrait : approche choisie

Pour satisfaire les exigences précédemment énoncées, il est utile de définir un nouveau type de raffinement. L'idée consiste à dire que de la même manière que l'on fournit des modèles, on en requiert d'autres. Un modèle fourni propose une solution « générique » a un problème et se sert, pour devenir opérationnel, de modèles requis décrivant seulement l'interface des modèles d'extension. Ainsi, le **raffinement par intégration de modèles requis** explicite la collaboration entre les modèles et leurs dépendances et permet de prendre en compte des informations dépendantes des plates-formes sans que le modèle raffiné soit dépendant des plates-formes. Pour cela, nous introduisons deux relations supplémentaires. La relation *requiring* entre deux modèles est une forme de relation de délégation indiquant qu'un modèle en utilise un autre pour réaliser une ou plusieurs tâches. La relation *requires* entre opérations met en correspondance une opération du modèle fourni avec des opérations de modèles requis pour expliciter la dépendance.

Les descriptions Picore suivantes décrivent comment raffiner successivement le modèle abstrait de Satin en intégrant les modèles requis correspondant à la prise en compte des contrats (`ContractExtension`) et des adaptations élémentaires (`AdaptationExtension`). Le premier raffinement `SatinRaffinement1` permet de décrire la dépendance d'un rôle et d'un port avec le système de contrat sous-jacent. L'opération `Role.isSubRoleOf` du modèle abstrait se réfère alors à la relation de conformité de la plate-forme cible à travers l'opération `ClassifierContract.conforms` du modèle requis de contrats. De même, l'opération `Port.isSubPortOf` du modèle abstrait se réfère à l'opération `FeatureContract.conforms` du modèle requis de contrats.

Notons que l'opération `ContractSystem.getContract` permet de retrouver un type de composant (resp. type fonctionnel) de la plate-forme à partir d'un nom de rôle (resp. signature d'un port). Comme le modèle raffiné se base sur la relation de conformité de la plate-forme sous-jacente, nous n'avons pas besoin de distinguer, au niveau des rôles, les ports synchrones ou asynchrones, obligatoires ou optionnels par exemple. La relation de conformité se charge de vérifier que les ports sont compatibles suivant règles de typage établies dans la plate-forme cible.

```
metamodel SatinBase
    class Port
    class Role
        attribute ports * Port
    class Component
       attribute roles * Role
    class AdaptationPattern
    ...

metamodel ContractExtension
    class Contract
        operation conforms(Contract c) : boolean
    class ClassifierContract extends Contract
    class FeatureContract extends Contract
    class ContractSystem
        operation getContract(String name) : Contract

metamodel SatinRaffinement1 refines SatinBase requiring ContractExtension
    class Role is refined
        operation isSubRoleOf(Role r) : boolean
                  requires ContractExtension::ClassifierContract.conforms,
                           ContractExtension::ContractSystem.getContract
```

```
class Port is refined
    operation isSubPortOf(Port p) : boolean
              requires ContractExtension::FeatureContract.conforms,
                       ContractExtension::ContractSystem.getContract
```

Le second raffinement SatinRaffinement2 permet de décrire la dépendance d'un schéma d'adaptation avec la représentation sous-jacente des adaptations dans la plate-forme cible. L'opération Role.filters du modèle abstrait destiné à vérifier si un composant peut jouer un rôle donné utilise également l'opération PointCutLanguage.match du modèle requis d'adaptations. La déduction des rôles d'un schéma via l'opération AdaptationPattern.deduceRolesFrom du modèle abstrait se fait par rapport aux ports utilisés ou adaptés dans les adaptations élémentaires décrites dans la plate-forme cible à travers les opérations ElementaryAdaptation.getProperties et PointCutLanguage.match du modèle requis d'adaptations.

La detection de cycles et de points de non déterminisme via les opérations AdaptationPattern.containsCycleFrom et AdaptationPattern.containsNDP du modèle abstrait utilisent aussi les opérations ElementaryAdaptation.getProperties et PointCutLanguage.match pour calculer l'ensemble des ports atteignables depuis chaque adaptation élémentaire. L'opération AdaptationPattern.containsNDP utilise en plus l'opération ElementaryAdaptation.withoutOrdering qui indique que l'adaptation considérée introduit de l'indéterminisme en choisissant de ne pas imposer d'ordre dans l'enchainement de certains ports utilisés.

Les deux autres opérations du modèle requis d'adaptations ElementaryAdaptation.getType et ElementaryAdaptation.isCompatibleWith sont utilisées dans l'expression des contraintes visant à assurer les propriétés de sûreté au niveau de la création (AdaptationPattern.createFrom) et l'application (AdaptationPattern.instantiate) de schémas d'adaptation (cf chapitre 7 pour plus de détail sur les contraintes). Ainsi, les schémas d'adaptation manipulent des adaptations élémentaires sans en connaître la représentation.

```
metamodel AdaptationExtension
   class PointCutLanguage
      operation match(String s1, String s2) : boolean
   class ElementaryAdaptation
      operation getPointcut() : String
      operation getProperties() : collection
      operation getType() : String in {'add', 'control'}
      operation withoutOrderingFor() : collection
      operation isCompatibleWith(ElementaryAdaptation ea) : boolean

metamodel SatinRaffinement2 refines SatinRaffinement1 requiring AdaptationExtension
   class Role is refined
      operation filters(Role r) : boolean
                requires AdaptationExtension::PointCutLanguage.match

   class AdaptationPattern is refined
      operation deduceRolesFrom(Vector adaptations) : Vector
                requires AdaptationExtension::ElementaryAdaptation.getProperties
      operation containsCycleFrom(Vector current, Vector visited) : boolean
                requires AdaptationExtension::PointCutLanguage.match,
                         AdaptationExtension::ElementaryAdaptation.getProperties
      operation containsNDP(Vector current, Vector visited) : boolean
                requires AdaptationExtension::PointCutLanguage.match,
                         AdaptationExtension::ElementaryAdaptation.getProperties,
                         AdaptationExtension::ElementaryAdaptation.withoutOrderingFor
```

```
operation createFrom(Vector roles, Vector adaptations)
          requires AdaptationExtension::ElementaryAdaptation.isCompatibleWith,
                   AdaptationExtension::ElementaryAdaptation.getType
operation instantiate(Vector components)
          requires AdaptationExtension::ElementaryAdaptation.isCompatibleWith
```

La figure 5.8 récapitule les différentes dépendances entre le modèle abstrait Satin et les modèles requis de contrats et d'adaptations.

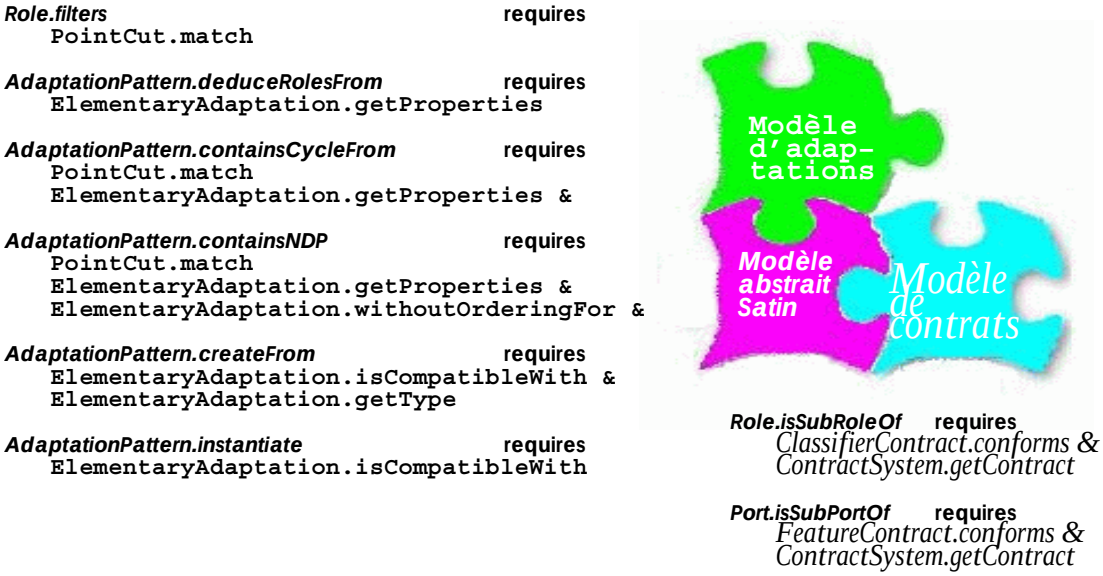

FIG. 5.8 – Puzzle du modèle abstrait Satin et des modèles requis de contrats et d'adaptations

Les modèles fournis et requis ne se projettent pas de la même manière : un modèle fourni propose de la valeur ajoutée qui doit être générée dans la plate-forme alors qu'un modèle requis sert seulement à localiser un point de collaboration avec la plate-forme. Seule l'interaction avec ce point doit être générée. Par ailleurs, le modèle requis est une forme de **contrat pour la projection**. Si le contrat est rempli (la plate-forme visée fournit un modèle compatible proposant des opérations équivalentes), on projette seulement les appels aux opérations requises par Satin et fournies par la plate-forme. Sinon, on raffine par substitution le modèle requis par un modèle fourni conforme qui projette les éléments manquants (y compris les opérations) à la plate-forme et on projette les appels aux opérations requises générées. Par exemple, le modèle requis `AdaptationExtension` a un certain nombre exigences. Si la plate-forme visée ne peut pas fournir un modèle compatible alors nous proposons d'utiliser notre propre modèle d'adaptations élémentaires `ASLExtension` basé sur le langage ASL décrit dans le chapitre 6.

La partie III destinée à montrer l'applicabilité du modèle Satin décrit la projection du modèle vers différentes plates-formes étudiées en 2. Cette partie a également pour but de montrer l'extensibilité du modèle, elle explique comment prendre en compte la sûreté de nouvelles formes d'adaptations liées aux composants nomades comme un raffinement simple du modèle abstrait `SatinBase`.

5.5.4 Avantages de l'approche « puzzle »

Le découpage en modèles fournis et modèles requis permet de mieux déterminer la frontière d'une modélisation. Il ne suffit pas de modéliser l'univers dans un but particulier mais aussi d'en délimiter le périmètre et de définir les conditions sous lesquelles la modélisation est utilisable. Ce découpage favorise donc la séparation des préoccupations au niveau des modèles et donc leur évolutivité. De ce point de vue, l'approche puzzle est dans la continuité des travaux autour de la construction de modèles à partir de modèles plus petits qui s'appuient sur des modèles paramétrés ou des calibres de modèles [36, 112, 81]. Une différence avec ces travaux est que l'approche puzzle tend à produire des modèles à un niveau de granularité plus important et correspondant vraiment à une préoccupation complète.

D'autre part, l'approche puzzle prépare également la collaboration des modèles tout comme l'apparition des ports requis sur les composants a facilité leurs compositions. Contrairement à [24] qui propose de composer les éléments des métamodèles non orthogonaux, l'approche puzzle réduit le problème de la composition de modèles à un problème de collaboration où les éléments sont mis en dépendance plutôt qu'en correspondance. Par exemple, nous avons travaillé sur cette problématique avec Satin et MicM qui sont deux modèles complémentaires : le premier travaillant autour de la sûreté des adaptations dynamiques d'application et le second contrôlant l'intégration statique de services. Cette étude a fait ressortir l'avantage à décrire clairement les deux modèles sous forme de puzzle pour en extraire les points de raccordement.

Enfin, l'approche puzzle est particulièrement adaptée aux modèles qui étendent les fonctionnalités des applications en ajoutant une nouvelle expertise ou une nouvelle compétence de manière générique quelque soit la technologie sous-jacente. Ce besoin est bien connu et traité dans le domaine des plates-formes à composants dont les conteneurs prennent en charge un certain nombre de services. Les services sont développés afin d'introduire de nouvelles compétences dans ces plates-formes sans impacter directement celles-ci ni les applications qu'elles hébergent. Une modélisation sous forme de puzzle permet d'identifier clairement, à travers l'utilisation de modèles requis, les points d'interaction entre les services (modèles fournis) et la plate-forme cible. Ainsi, l'approche puzzle est un avantage pour modéliser de tels services et pour faciliter leur intégration dans des plates-formes en fournissant simplement des modèles compatibles avec les modèles requis. L'approche puzzle couplée à l'approche « service » est donc un atout pour les modèles exécutables tel que le modèle Satin : cela permet de faciliter l'étape de mise en œuvre de ce type de modèles en les rendant opérationnels sous forme d'un service (voir chapitre 8).

« Un concept est une invention à laquelle rien ne correspond exactement, mais à laquelle nombre de choses ressemblent. »

Friedrich Nietzsche

« Nous avons, en effet, l'habitude de poser une certaine Forme et une seule, pour chaque groupe d'objets multiples auxquels nous donnons le même nom. »

Platon

6 Rôles et schémas d'adaptation

Ce chapitre est consacré aux deux concepts pivots du modèle Satin : les rôles, une forme de typage adaptée à la notion d'évolution et les schémas d'adaptation, une forme de réification des opérations d'adaptation. Ces deux concepts sont étroitement liés, les schémas étant typés par les rôles et les rôles « mémorisant » les adaptations appliquées aux composants à travers les schémas.

La section 6.1 détaille plus précisément le contenu d'un rôle et en précise les caractéristiques. La section 6.2 revient sur le concept de schéma d'adaptations et donne quelques propriétés sur l'utilisation de ces schémas. La section 6.3 discute des choix de modélisation des rôles et des schémas d'adaptation qui ont été faits.

6.1 Les rôles : un typage pour l'adaptation

Le typage dans les langages orientés Objet permet d'assurer que les objets, de par la définition statique de leur type, sauront répondre aux messages prévus. Or, nous avons vu dans le chapitre 3 que cette notion de type n'est pas adaptée à une vision évolutive du typage qui est pourtant une propriété essentielle dès lors que l'on s'intéresse à des applications où les entités sont interconnectées, déconnectées et adaptées dynamiquement. Ce chapitre présente une forme de typage adapté au concept d'applications évolutives.

Dans la section 6.1.1 nous revenons sur le besoin d'étendre la notion de type et les diverses extensions qui ont été faites dans la littérature. Les sections suivantes détaillent plus précisément le concept de rôle et en précisent les caractéristiques.

6.1.1 Des travaux autour de l'extension du type

Le mécanisme de typage sert de base à la vérification de la correction des programmes et à la substitution des objets. La plupart des applications ne se base que sur le *contrat syntaxique*, la forme la plus répandue de typage qui ne prend en compte que les noms des méthodes et le type des paramètres pour effectuer les vérifications. Cependant, de nombreux travaux étendent ce mécanisme de typage principalement pour répondre à la problématique de composition. Ainsi,

l'évolution des objets vers les composants met en avant la volonté de rendre explicite toute interaction avec ou à partir du type de composant. Un type de composant[1] ne se limite alors plus à définir les fonctionnalités qu'il offre (approche usuelle des interfaces orientées objet), mais aussi les fonctionnalités qu'il utilise qui jusqu'ici n'apparaissaient pas, tendant ainsi à étendre la notion de type [72, 86].

Quant aux *contrats comportementaux* [78, 31, 39] ainsi que les travaux de [85], ils étendent le type en ajoutant des contraintes visant à conditionner la composition de composants. Ces contrats comportementaux sont constitués d'un ensemble de propriétés généralement exprimées sous forme de pré-conditions, de post-conditions, et d'invariants liés à une méthode. De même, les *contrats de synchronisation* [3, 82] décorent les types par des protocoles indiquant la façon d'utiliser un composant (enchaînements de services autorisés) et les interactions des composants. Il s'agit de prendre en compte des interactions du composant avec l'extérieur, mais ayant une influence sur son comportement interne. On peut décrire leur exécution sous forme de suite d'étapes. L'indéterminisme des appels et la concurrence de l'exécution des opérations est à prendre en compte dans la description de ces protocoles. Enfin, les *contrats de qualité de service* [38] ajoutent au type des propriétés de performance, de maintenabilité, portabilité ou encore de disponibilité afin d'indiquer comment rendre ou requérir un service. Le temps de réponse ou la précision du résultat sont des propriétés qui entrent dans cette catégorie de contrats.

Elargir la notion de type pour vérifier les propriétés de sûreté nous semble être également une voie intéressante pour mieux gérer l'évolution des composants par adaptation. Le type d'un composant n'est alors plus restreint à un ensemble de messages auxquels il sait répondre, mais aussi aux « types » prérequis des composants avec lesquels il est assemblé, à des contrôles sur ses méthodes (nous détaillons dans la section 6.2 de ce chapitre la notion de contrôle) et à des *contrats d'adaptation* (nous détaillons dans le chapitre 7 ces contrats). Cette extension de type est utilisée pour garantir la sûreté de fonctionnement des composants lorsqu'ils sont adaptés à l'exécution. Pour cette raison et afin d'éviter toute confusion avec l'approche classique du typage, nous avons désigné cette forme de typage pour l'adaptation par le terme « rôle ». Dans le chapitre 5, nous avons ainsi vu que les rôles étaient utilisés pour évaluer si une adaptation ne remettait pas en cause la sûreté de fonctionnement d'une application en déterminant l'applicabilité de celle-ci sous la forme d'un schéma d'adaptation.

Dans la section suivante, nous détaillons plus précisément le contenu d'un rôle pour en extraire la valeur ajoutée vis-à-vis des types classiques.

6.1.2 Rôles primitifs, génériques, abstraits et concrets

Il existe quatre catégories de rôles (figure 6.1) :
- les **rôles primitifs** représentent les types primitifs,
- les **rôles génériques** sont attachés aux schémas d'adaptation et spécifient les propriétés attendues des composants pour qu'une adaptation soit applicable,
- les **rôles abstraits** représentent les types formels (statiques) c'est à dire les types attendus dans la signature des méthodes de l'application modélisée,
- les **rôles concrets** sont en quelque sorte des types effectifs car ils sont attachés aux composants tout au long de leur cycle de vie (ils « mémorisent » les adaptations subies pas les composants) et servent à vérifier l'applicabilité d'un schéma d'adaptation à un composant.

Notons que les rôles abstraits sont déduits des applications alors que les rôles génériques sont déduits des adaptations élémentaires. Enfin les rôles concrets sont calculés à partir des rôles abstraits et génériques. Seuls les rôles concrets évoluent dans le temps.

[1] "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only." [113]

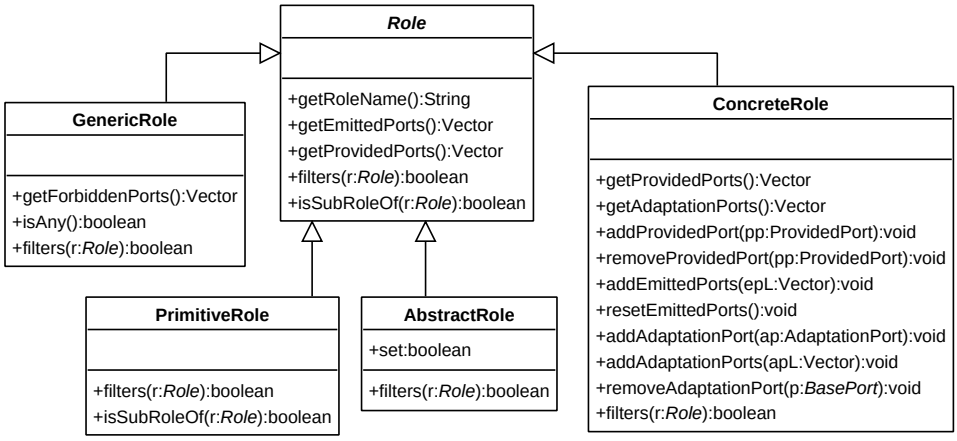

FIG. 6.1 – Hiérarchie UML des roles

Les rôles sont constitués de ports (figure 6.2). Comme les rôles, il existe différentes catégories de ports : **port fourni**, **port émis**, **port interdit** et **port d'adaptation**.

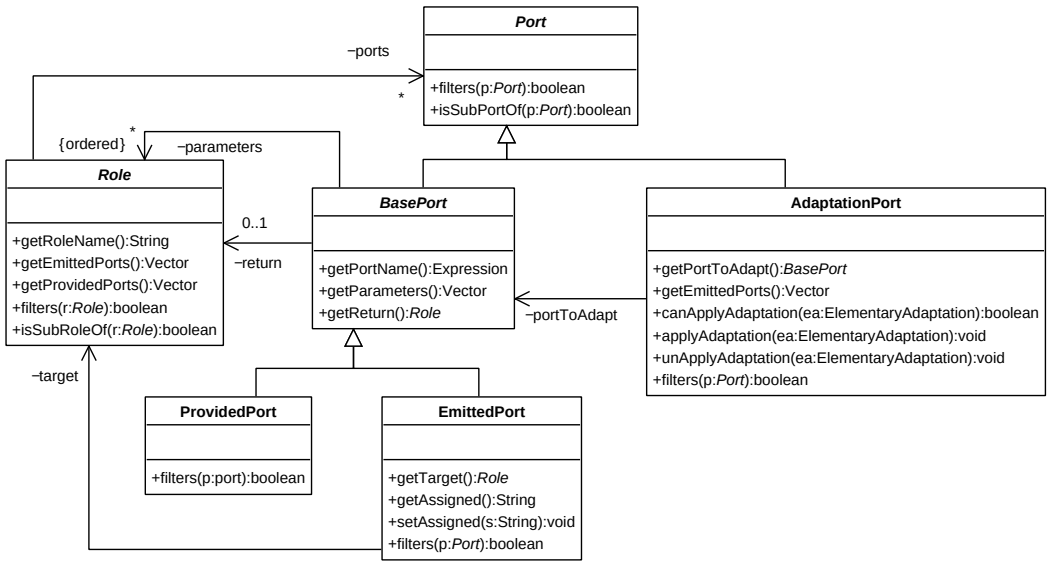

FIG. 6.2 – Hiérarchie UML des ports

La signification du port change suivant la catégorie de rôle auquel ils sont associés :
- Un rôle générique associé à un schéma d'adaptation *sa* est constitué de :
  - ports fournis, chacun représentant un ensemble de fonctionnalités qui doivent être offertes par le composant auquel on applique le schéma *sa*,
  - ports émis, chacun représentant une fonctionnalité requise par une adaptation
  - ports interdits, chacun représentant une fonctionnalité à ajouter à un composant et qui ne doit donc pas être déjà présente au niveau de ce composant.
- Un rôle concret associé à un composant *c* est constitué de :
  - ports fournis, chacun étant un point d'accès à une fonctionnalité effectivement offerte par le composant *c*,

  - ports émis, chacun représentant une fonctionnalité utilisée par le composant c dans le cadre d'une adaptation,
  - ports d'adaptation, chacun étant un port fourni ou émis qui a été adapté. Un port d'adaptation permet de décrire les modifications de comportement associées aux ports fournis et émis du composant c et de décrire les assemblages dynamiques de composants. En effet, les ports d'adaptation expriment les relations entre ce qui est fourni par le composant et ce qui est requis par le composant pour une adaptation.

- Un rôle abstrait est constitué de ports fournis, chacun représentant une fonctionnalité offerte attendue lors de la substitution de composants.

✎ Nous utilisons les notations suivantes pour caractériser les composants, rôles et ports :

Composant :	rôles :	*soit* $\Gamma(c)$, *l'ensemble des rôles concrets associés au composant c*
Role :	type :	*soit* $\mathcal{RG}$, *l'ensemble des rôles génériques,* $\mathcal{RA}$, *l'ensemble des rôles abstraits,* $\mathcal{RC}$, *l'ensemble des rôles concrets de tous les composants et* $\mathcal{RP}$, *l'ensemble des rôles primitifs*
	élément neutre :	*soit* $Any \in \mathcal{RG}$, *la racine des rôles génériques, il désigne n'importe quel composant*
	ports fournis :	*soit* $F(r)$, *l'ensemble des ports fournis du rôle* $r \in \mathcal{RG} \cup \mathcal{RA} \cup \mathcal{RC}$
	ports émis :	*soit* $E(r)$, *l'ensemble des ports émis de* $r \in \mathcal{RG} \cup \mathcal{RC}$
	ports interdits :	*soit* $I(r)$, *l'ensemble des ports interdits de* $r \in \mathcal{RG}$
	ports d'adaptation :	*soit* $A(r)$, *l'ensemble des ports d'adaptation de* $r \in \mathcal{RC}$
Port :	type :	*soit* $\mathcal{PF}$, *l'ensemble des ports fournis,* $\mathcal{PE}$, *l'ensemble des ports émis,* $\mathcal{PI}$, *l'ensemble des ports interdits et* $\mathcal{PA}$, *l'ensemble des ports d'adaptation*
	nom :	*soit* $name(p)$, *le nom associé au port* $p \in \mathcal{PF} \cup \mathcal{PE} \cup \mathcal{PI} \cup \mathcal{PA}$
	paramètres :	*soit* $args(p)$, *l'ensemble des rôles paramètres de* $p \in \mathcal{PF} \cup \mathcal{PE} \cup \mathcal{PI} \cup \mathcal{PA}$
	retour :	*return*(p), *le rôle de retour de* $p \in \mathcal{PF} \cup \mathcal{PE} \cup \mathcal{PI} \cup \mathcal{PA}$ *return*(p) *est un ensemble singleton ou vide*
	cible :	*soit* $target(ep)$, *le rôle associé au port émis* $ep \in \mathcal{PE}$
	adaptation :	*soit* $adaptation(ap)$, *l'adaptation élémentaire associée au port d'adaptation* $ap \in \mathcal{PA}$

Revenons à l'exemple des agendas introduit dans le chapitre 5. On a :

$\Gamma(agendaEquipe) = \{AgendaDeBase\}$

$F(AgendaDeBase) = \{\texttt{addRdv(Rdv)}, \texttt{removeRdv(Rdv)}, \texttt{getRdv(Date) : Rdv}\}$

$E(AgendaDeBase) = \{\}$

$A(AgendaDeBase) = \{\}$

Notons que les paramètres de port `Rdv` et `Date` sont des rôles abstraits. `AgendaSynchronisant` et `AgendaSynchronisé` sont les rôles génériques déduits du schéma d'adaptation `Synchronisation`. Le paramètre de port `Meeting` est également un rôle générique qui équivaut à `Any` puisque aucune contrainte ne concerne ce rôle dans le schéma `Synchronisation`. L'application du schéma respectivement à *agendaAM* en tant que agenda synchronisé et à *agendaEquipe* en tant que agenda synchronisant a pour conséquence de modifier le rôle de *agendaEquipe* comme le montre la figure 6.3. Par contre, le rôle de *agendaAM* ne change pas car aucune adaptation élémentaire du schéma ne le concerne directement, ce composant est seulement utilisé dans cette adaptation.

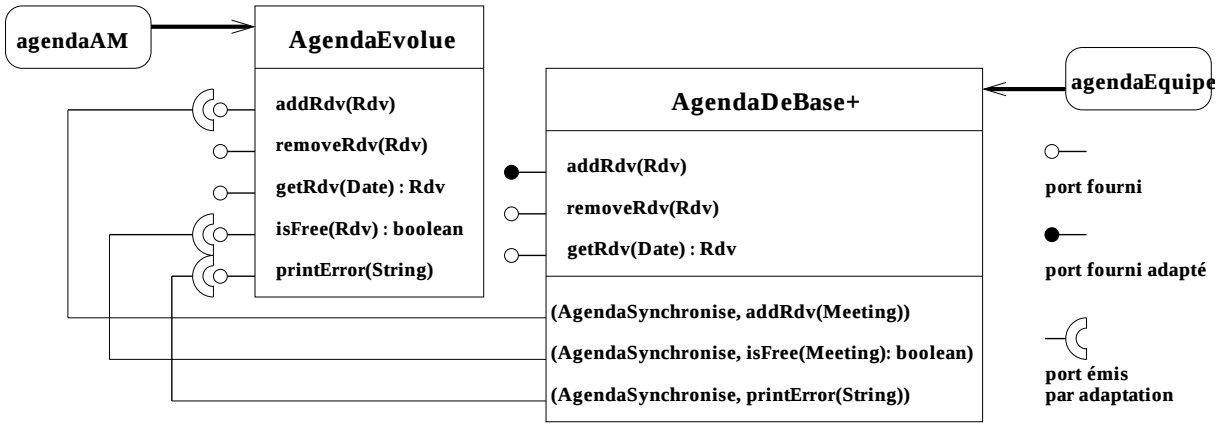

FIG. 6.3 – Résultat de l'application du schéma `Synchronisation` à *agendaAM* et à *agendaEquipe*

Les ports émis vers le composant jouant le rôle de l'agenda synchronisé (en l'occurrence *agendaAM*) sont ajoutés au rôle de *agendaEquipe*. Un port d'adaptation correspondant au port ayant subi une adaptation (en l'occurrence `addRdv`) est également créé (parce qu'il s'agit de la première fois que le port est adapté) et « mémorise » l'adaptation. Après l'application du schéma on a donc :

$\Gamma(agendaEquipe) = \{AgendaDeBase+\}$

$F(AgendaDeBase+) = \{\texttt{addRdv(Rdv)}, \texttt{removeRdv(Rdv)}, \texttt{getRdv(Date) : Rdv}\}$

$E(AgendaDeBase+) = \{$ (`AgendaSynchronise`, `addRdv(Rdv)`),
(`AgendaSynchronise`, `isFree(Rdv) :boolean`),
(`AgendaSynchronise`, `printError(String)`)$\}$

$A(AgendaDeBase+) = \{$ (`addRdv(Rdv)`, `adaptation`)$\}$

Notons que si d'autres composants créés à partir du même template que *agendaEquipe* et donc comportant un rôle concret *AgendaDeBase* ne sont pas adaptés comme *agendaEquipe*, alors leur rôle n'est pas modifié.

6.1.3 Relations de conformité sur les rôles : Positionnement par rapport au modèle requis de contrats

Selon Wegner [121], il existe deux sortes de relations de sous-typage. Le sous-typage faible ou sous-typage d'interface (conformité structurelle basée sur le contrat syntaxique) implique que « t' est sous type de t si une instance de type t' peut être substituée quand une instance de type t est attendue ». Le sous-typage fort ou sous-typage comportemental (conformité basé sur le contrat comportemental, de synchronisation ou de qualité de services) implique que « pour un système S, t' est sous type de t si une instance de type t' peut être substituée quand une instance de type t est attendue et le comportement de S est inchangé ». Liskov [67] complète la notion de sous-typage fort par la condition suivante : « Soit P(x) une propriété vraie sur les objets x de type t, alors P(y) doit être vraie pour les objets y de type t' où t' est un sous type de t ».

Satin travaillant sur une métareprésentation de l'application, la gestion de la conformité d'un composant vis-à-vis d'un rôle est double. Le typage du modèle Satin utilise deux notions de conformité : la substituabilité et le filtrage. Les deux prochaines sections montrent comment ces deux relations de conformité se positionnent par rapport aux relations de sous-typage.

La substituabilité : Conformité entre rôles abstraits et concrets

La **substituabilité** est une relation de conformité entre rôles abstraits et concrets. La substituabilité est utilisée, à travers l'opération `Role.isSubRoleOf` du modèle abstrait, pour résoudre les contraintes d'interdépendances (voir la section 6.2.4 sur les rôles ancrés pour plus de détails) entre les rôles de différents composants auxquels on souhaite appliquer un schéma d'adaptation et pour déterminer si un composant peut en remplacer un autre. Dans l'exemple des agendas, le rôle générique `Meeting` correspond à `Any`. Utilisé en tant que paramètre de plusieurs ports de différents rôles (`AgendaSynchronisant.addRdv`, `AgendaSynchronise.addRdv` et `AgendaSynchronise.isFree`), des contraintes d'interdépendances doivent être respectées pour assurer un typage correct.

Certains ADLs tels que ArchJava [5] abstraient la notion de connecteur pour permettre la définition « customisée » du typage. Dans notre cas, la relation de substituabilité dépend des règles de sous-typage utilisées dans la plate-forme sous-jacente. Ainsi, la tâche de déterminer s'il y a substituabilité ou non est délèguée à la plate-forme cible via l'opération `ClassifierContract.conforms` du modèle requis de contrats `ContractExtension` défini en 5.5. Généralement, la substituabilité est basée sur un sous-typage faible :

- conformité sur les noms (typage nominal),
- conformité au sens de l'héritage (sous-classage),
- conformité au sens du polymorphisme d'inclusion.

Dans les deux premiers cas, on n'a pas besoin de « comparer » les ports pour déterminer s'il y a substituabilité ou non. Il en est de même, pour les rôles correspondant à des types primitifs. A l'inverse, la substituabilité au sens du polymorphisme d'inclusion est fonction de la conformité entre ports. Dans ce cas, la relation de substituabilité sur les rôles, notée $r' \leq r$, est définie en fonction de la relation de substituabilité sur les ports, notée $p' \leq p$, et inversement (les deux relations sont interdépendantes). Par conséquent, la relation de substituabilité sur les ports est satisfaite quand la relation de sous-typage sur les méthodes de la plate-forme ciblée est vraie (opération `FeatureContract.conforms` du modèle requis de contrats).

Definition 1 (Substituabilité - non nominale - sur les rôles)

$\forall\, r \in \mathcal{RA} \cup \mathcal{RC} \cup \mathcal{RP} \mid r \leq r$ *(reflexivité)*

$\forall\, r, r' \in \mathcal{RA} \cup \mathcal{RC} \mid r' \leq r \nLeftrightarrow r \leq r'$ *(antisymétrie)*

$\forall\, r, r' \in \mathcal{RP} \mid r' \leq r \nLeftrightarrow r \leq r'$ *(antisymétrie)*

$\forall\, r1, r2, r3 \in \mathcal{RA} \cup \mathcal{RC} \cup \mathcal{RP} \mid r1 \leq r2$ *and* $r2 \leq r3 \Rightarrow\ r1 \leq r3$ *(transitivité)*

$\forall\, r \in \mathcal{RA} \cup \mathcal{RC}, r' \in \mathcal{RP} \mid r' \not\leq r$

$\forall\, r \in \mathcal{RP}, r' \in \mathcal{RA} \cup \mathcal{RC} \mid r' \not\leq r$

Definition 2 (Substituabilité - au sens du polymorphisme d'inclusion - sur les rôles)

$\forall\, r, r' \in \mathcal{RA} \cup \mathcal{RC} \mid r' \leq r \Leftrightarrow$

$\forall\, p1 \in F(r), \exists\, p1' \in F(r') \mid p1' \leq\ p1$ *and*

$\forall\, p2' \in E(r'), \exists\, p2 \in E(r) \mid p2' \leq\ p2$

Notons que la relation de substituabilité peut également se baser sur un sous-typage fort si la plate-forme ciblée utilise des contrats « non syntaxiques » (contrats comportementaux, de synchronisation ou de qualité de service). Ainsi, deux ports qui matchent structurellement peuvent ne pas être choisis s'ils ne sont pas compatibles par rapport à leurs contrats non syntaxiques.

Le filtrage : Conformité vis-à-vis des rôles génériques

Le **filtrage** est une relation de conformité entre rôles génériques ou primitifs, d'une part, et rôles abstraits ou concrets ou primitifs, d'autre part. Le filtrage est utilisé, à travers l'opération `Role.filters` du modèle abstrait, pour vérifier si un composant est capable de jouer le rôle que l'on veut lui attribuer lorsqu'on applique un schéma d'adaptations[2]. Le filtrage est basé sur la correspondance de coupes - opération `match` du modèle d'adaptations élémentaires `AdaptationExtension` (cf. section 5.5)[3]. Notons que nous sous-utilisons le modèle d'adaptations élémentaires en ce qui concernent les coupes dans le sens où nous nous servons de la correspondance de coupes au niveau des noms de méthodes mais pas pour les noms de rôles dans la mesure où les noms de rôles génériques déduits des adaptations élémentaires n'ont pas de lien avec l'application.

La relation de filtrage sur les rôles, notée $r' <\# r$, est plus faible qu'une relation de sous-typage. Elle n'a pas pour but de garantir la substituabilité mais d'assurer que toutes les propriétés minimales attendues pour une adaptation sont présentes. Contrairement à la substituabilité, le filtrage est défini indépendamment des plates-formes. La relation de filtrage sur les rôles se définit en fonction de la relation de filtrage sur les ports, notée $p' <\# p$ et inversement (les deux relations sont interdépendantes).

Definition 3 (Filtrage sur les rôles)

$\forall\, r' \in \mathcal{RA} \cup \mathcal{RC} \cup \mathcal{RP} \mid r' <\# Any$

$\forall\, r \in \mathcal{RG} - \{Any\}, r' \in \mathcal{RP} \mid r' \not<\# r$

$\forall\, r \in \mathcal{RP}, r' \in \mathcal{RP} \mid r' <\# r \Leftrightarrow\ r' \leq r$

$\forall\, r \in \mathcal{RG} - \{Any\}, r' \in \mathcal{RA} \cup \mathcal{RC} \mid r' <\# r \Leftrightarrow$

$\forall\, p1 \in F(r), \exists\, p1' \in F(r') \mid p1' <\# p1$ *and*

$\forall\, p2 \in E(r), \exists\, p2' \in E(r') \mid p2' <\# p2$ *and*

$\forall\, p3 \in I(r), \nexists\, p3' \in F(r') \mid p3' <\# p3$

[2] Cette relation est notamment utilisée pour vérifier qu'un port à ajouter n'est pas présent dans le composant avant adaptation.

[3] Par exemple, le nom de méthode `getX` correspond à `get*` dans un modèle d'adaptations élémentaires où la notion de coupe est basée sur un langage incluant l'utilisation d'expressions régulières sur les noms de méthodes.

Notons que la conformité au sens du filtrage est inversée pour les ports émis par rapport la conformité au sens de la substituabilité.

Definition 4 (Filtrage sur les ports)

$p' <\# p \Leftrightarrow$

$name(p') \subseteq name(p)$ *(l'opérateur* $\subseteq$ *correspond ici à l'opération* `match`*) and*

$card(args(p)) = card(args(p'))$ *and*

$\forall\ rp_i \in args(p)$ *and* $rp'_i \in args(p')$ |

$rp_i \in \mathcal{RG}$ *and* $rp'_i \in \mathcal{RA} \cup \mathcal{RC} \cup \mathcal{RP} \Rightarrow\ rp'_i <\# rp_i$ *(covariance) and*

$rp_i \in \mathcal{RP}$ *and* $rp'_i \in \mathcal{RP} \Rightarrow\ rp_i <\# rp'_i$ *(contravariance) and*

$(\exists\ rr$ *in* $return(p)) \Rightarrow (\exists\ rr' \in return(p')$ *and* $rr' <\# rr)$ *(covariance)*

La nécessité de covariance sur les paramètres de port non primitifs est démontrée de la façon suivante. Soit l'adaptation élémentaire :
`modifyPort a.m1(B b) -> b.n1 () ; b.n2()`

Soit `A`, le rôle générique de `a` et `B`, le rôle générique de `b`, déduits de l'adaptation élémentaire tel que :

$F(A) = \{$`m1(B)`$\}$

$F(B) = \{$`n1(),n2()`$\}$

Soient les composants `c` et `c'` tels que :

$\Gamma(c) = \{$`C`$\}$

$\Gamma(c') = \{$`B'`$\}$

Supposons que l'adaptation élémentaire soit appliquée à `c`. Pour que l'adaptation élémentaire soit applicable du point de vue du typage, il faut :

`C` <# `A`

Ce qui implique qu'il existe un port `m1(B')` tel que :

`m1(B')` $\in F(C)$

A l'appel de `c.m1(c'),n1()` et `n2()` vont être appelées sur `c'`. Ceci implique que `B'` comporte au moins les ports `n1()` et `n2()`. **Donc il faut** `B'` **<#** `B` **pour avoir** `C` **<#** `A`.

6.1.4 Spécificités des rôles par rapport au typage classique

Les rôles définis par le modèle Satin se différencient des types classiques (contrats syntaxiques) dans le sens où ils embarquent plus que les fonctionnalités fournies et requises, approche traditionnelle du typage de composants. A l'instar des différentes déclinaisons de types qui incluent contrats comportementaux (pre / post conditions, invariants), contrats / protocoles de synchronisation (automates de Büchi, réseaux de Petri, ...), et contrats de qualité de service pour affiner la composition de composants, les rôles du modèle Satin sont enrichis par des informations visant à répondre à la problématique de sûreté face aux adaptations.

Ainsi, les rôles n'incluent pas les fonctionnalités requises de base ou encore les propriétés liées aux contrats comportementaux, de synchronisation ou de qualité de service. Quelque soient les propriétés incluses dans les types, ce qui nous intéresse de modéliser est uniquement les prédicats permettant de déterminer si un composant peut jouer un rôle donné au sens du filtrage d'une part et au sens de la substituabilité d'autre part afin de déterminer si un schéma d'adaptation est applicable à un composant donné. Or, nous n'avons besoin que de la signature des ports fournis (fonctionnalités offertes) comme information sur le type des composants pour déduire la compatibilité de deux rôles que ce soit au sens du filtrage comme de la substituabilité (toutes les autres informations des rôles étant construites au fur et à mesure en fonction des adaptations). Par conséquent, quelque soit la forme de typage utilisée dans la plate-forme cible (structurel, comportemental, ...), ceci n'a aucune influence sur le modèle des rôles.

Dans la suite sont exposées quatre caractéristiques des rôles du modèle Satin afin de les comparer aux types usuels.

Typage mutatif

Les rôles associés aux composants (rôles concrets) peuvent évoluer dans le temps. Ils ne doivent pas être confondus avec les types classiques car ils supportent des propriétés dynamiques : ils décrivent en temps réel les différentes adaptations effectuées sur leur composant respectif. Contrairement aux rôles définis par Peschanski [98], les rôles concrets ne sont pas des métaobjets dans le sens où ils n'ont pas pour vocation de contrôler la structure et le comportement des composants.

Tous les composants issus d'un même template ont initialement des rôles équivalents. Ensuite les rôles de chaque composant évoluent indépendamment les uns des autres à chaque adaptation en respectant une certaine cohérence. Ainsi, deux composants issus d'une même fabrique peuvent avoir des rôles différents dans le temps si on leur a appliqué des schémas d'adaptation différents. Comme un rôle concret est attaché à un composant et non pas à un ensemble de composants, un rôle concret est plus proche de la notion de prototypes ou d'acteurs que de la notion de classe.

Typage implicite et local

Dans les approches classiques de typage statique, tous les types utilisés doivent être spécifiés et déclarés dès le départ afin de déterminer à la compilation la correction d'un programme. C'est-à-dire que pour utiliser un composant, on doit le déclarer comme étant d'un certain type T définissant, au minimum, l'ensemble des fonctionnalités auxquelles le composant sait répondre où T doit être explicitement décrit par l'utilisateur. D'autre part, un type T déclaré ne peut se référer qu'à une seule définition. A l'inverse, les rôles génériques sont déduits des informations liées aux adaptations élémentaires d'un schéma d'adaptation (un rôle est local à un schéma). Ils ne sont donc pas construits à l'avance mais déduits à la demande et seulement en fonction des besoins d'adaptation à un instant précis. Un rôle générique correspond aux contraintes minimales qu'un composant doit satisfaire pour l'adaptation en question.

Si l'on suivait une approche classique de typage statique, les rôles génériques des schémas devant être définis à l'avance, ils contiendraient par conséquent trop d'exigences. L'aspect déclaratif et global des rôles se traduirait alors par un rétrécissement de l'ensemble des composants auxquels un schémas d'adaptation est applicable. En effet, certains composants ne satisfaisant pas une certaine contrainte pourtant non nécessaire à l'adaptation à traiter seraient exclus de cet ensemble. A l'inverse, notre choix pour l'aspect non déclaratif et local des rôles fait que les composants sont contraints au minimum. Par exemple, deux rôles génériques de même nom[4] appartenant à deux schémas d'adaptations différents ne sont pas forcément équivalents dans notre

[4]Notons que le nommage des rôles par l'utilisateur dans la définition des schémas d'adaptation ne présente pas de sémantique en soi mais clarifie pour celui-ci à la fois le « rôle » que joue le composant et permet également en cas de violation de propriété un retour d'information plus explicite pour localiser l'erreur.

approche puisqu'ils ne sont déduits qu'à partir des informations d'un seul schéma. L'applicabilité d'un schéma d'adaptation est ainsi plus souvent accordée (les rôles étant moins contraints) sans pour autant affaiblir la sûreté de l'adaptation.

Typage générique

Les méthodes template de C++ ou les classes génériques de Java 1.5 représentent une forme de généricité dans le sens où elles permettent d'effectuer un traitement sur un ensemble de données sans fixer leur type. Cette forme de généricité est toute fois limitée, elle permet simplement de factoriser du code commun pour un ensemble de données indépendamment de leur type mais ne permet pas de désigner plusieurs types à la fois. Nous nommons cette forme de généricité dans la suite comme étant une généricité de conteneurs.

Les rôles génériques (déduits des adaptations élémentaires d'un schéma) expriment les propriétés structurelles que les composants associés à une adaptation doivent exhiber pour que l'adaptation soit possible de la même façon que les interfaces de collaboration de même nom définies dans UML [91] ou que les rôles des travaux sur le « role modeling » [9] définissent les critères que les composants doivent remplir pour pouvoir participer à une collaboration sans dénoter une classe en particulier. Contrairement à la généricité de conteneurs, cette forme de généricité permet donc de désigner un ensemble d'objets aux caractéristiques communes mais au type différent. Par exemple, un rôle générique comportant un port dont la signature est `store()` correspond à l'ensemble des composants comportant au moins un port conforme à ce port. Ainsi deux composants de classes d'implémentation différentes mais comportant un port conforme à `store` peuvent tous les deux jouer le rôle caractérisé par ce port indépendamment du fait qu'ils aient un type différent dans la plate-forme.

Les rôles génériques introduisent une seconde forme de généricité dans le sens où ils se basent sur le modèle de coupes pour caractériser les ports. Par exemple, dans le cas où le modèle de coupe utilisé est basé sur l'utilisation d'expressions régulières, un rôle générique comportant un port dont la signature est `get*() : Any` correspond à l'ensemble des composants fournissant au moins un accesseur en lecture. On retrouve cette notion de généricité dans les langages d'aspects toujours au travers de cette notion de coupe [59, 97]. Cette forme de généricité découle de la relation de conformité basée sur le filtrage. Les adaptations élémentaires peuvent être appliquées aux composants sans se baser sur un « matching » exact des noms de méthodes mais sur un matching au sens des expressions régulières par exemple et, encore une fois, le minimum doit être vérifier puisqu'on ne se base pas par rapport à des besoins antérieurs.

Typage et accès

La décomposition des implantations en différents rôles qui représentent les facettes d'un composant dépend d'un ensemble d'interfaces spécifiées pour le template. Ainsi, un template peut être défini de trois façons :

- soit à partir d'une implantation seule (cas des plates-formes qui permettent de s'adresser directement à un composant et donc à l'ensemble de ses fonctionnalités sans passer par des interfaces d'accès [17, 97])
- soit à partir d'une ou plusieurs d'implantations et d'un ensemble de rôles représentant les différentes interfaces d'accès (facettes) du composant (cas des plates-formes imposant de s'adresser à un composant qu'à partir de ses facettes [101, 86] (l'utilisation de plusieurs implantations représente le cas où le composant est fragmenté et propose des points de vue différents sur certaines fonctionnalités [89]).
- soit à partir d'une implantation et d'un rôle représentant des besoins en terme d'adaptation pour créer des composants à la volée au moment où ils sont adaptés pour la première fois.

Certaines plates-formes se basent sur des systèmes de types où un composant ne peut être utilisé qu'en fonction d'une facette (CCM, Fractal, Sofa, NOAH / EJB, ...). Ceci implique de jongler entre différentes facettes lorsque les fonctionnalités à utiliser sont définies dans différentes interfaces. Mais d'autres plates-formes permettent de s'adresser directement à un composant afin d'utiliser n'importe quelle fonctionnalité du composant même si celui-ci implémente plusieurs interfaces (JAC, Compose*, NOAH / Local, ...). Dans le modèle Satin, un composant peut être utilisé (et donc adapté) pour l'union d'un sous-ensemble des rôles qu'il propose ou seulement vis-à-vis d'un seul rôle suivant que la propriété P_2 sur la garantie de visiblité doit être vérifiée ou non. Les règles de conformité au sens du filtrage (voir section 6.1.3) ne changent pas. Par contre, on ne prend pas en compte le même ensemble de rôles pour vérifier s'il y a conformité ou non.

- Si P_2 doit être vérifier, il faut qu'il existe un rôle du composant qui soit conforme au rôle que doit jouer le composant dans l'adaptation.
- Sinon, il faut qu'il existe un sous-ensemble de rôles $\mathcal{R}$ du composant tel que le rôle constitué de l'union des ports des rôles de ce sous-ensemble $\mathcal{R}$ soit conforme au rôle que doit jouer le composant dans l'adaptation.

Par exemple, supposons :

$\Gamma(agendaAM)$ = {`ConsultationAgenda`, `GestionAgenda`}

$F(ConsultationAgenda)$ = {`getRdv`}

$F(GestionAgenda)$ = {`addRdv` , `removeRdv`}

Une adaptation telle la synchronisation que nous avons vu dans le chapitre 5 où l'agenda doit fournir à la fois `addRdv`, `removeRdv`, et `getRdv` n'est pas possible dans une plate-forme comme OpenCCM. Dans ce cas, l'opération `filters` se base que sur un seul rôle du composant. Or, il n'existe aucun rôle fournissant les trois ports requis par l'adaptation. A l'inverse, cette adaptation est possible dans une plate-forme comme JAC car il est possible d'accéder des ports appartenant à des facettes différentes directement par le composant (i.e. l'opération `filters` se base sur le rôle résultant de l'union des rôles du composant).

6.2 Schémas d'adaptations : Unités d'adaptations

La section 6.2.1 présente une concrétisation particulière du modèle d'adaptations élémentaires : `ASLExtension`. La section 6.2.2 décrit les étapes nécessaires à la concrétisation de tout modèle d'adaptations élémentaires. La section 6.2.3 démontre l'utilité des schémas d'adaptations comme unité d'application et de réutilisation des adaptations élémentaires. Enfin, la section 6.2.4 termine sur le typage des adaptations élémentaires.

6.2.1 ASL : langage de concrétisation du modèle d'adaptations élémentaires

Pour illustrer les exemples des chapitres 8, 7 et 5, nous nous sommes appuyés sur une concrétisation particulière du modèle requis d'adaptations élémentaires `AdaptationExtension`. Nous utilisons le langage ASL (Adaptation Specification Language) pour exprimer les adaptations. Cette section détaille ce langage et son modèle, `ASLExtension`, et évalue ses possibilités en terme de pouvoir d'expression des adaptations et en terme de résolution des conflits de composition.

Les contrôles : déclinaison des adaptations élémentaires

Dans le chapitre 2, nous avons vu qu'un certain nombre de plates-formes permettent de modifier le comportement des fonctionnalités de composants. Pour mettre en œuvre ce type d'adaptation élémentaire, des contrôles sont implicitement ajoutés aux composants. La forme de contrôle indique généralement la nature de l'impact exercé sur le composant ou le « lieu » de cet impact.

Par exemple, les filtres de composition [13] définissent un certain nombre de types de filtres qui correspondent à différentes formes de contrôle exercés sur le composant adapté. Les types de filtres de composition donnent des informations sur la façon dont le comportement d'une fonctionnalité est modifié. Il en est de même des opérateurs ISL de Noah [17]. Quant aux contrôles du langage AspectJ [59], ils indiquent le lieu du contrôle : `before`, `around` et `after`. Enfin, pour les approches telles que JAC [97] et FAC [100] où aucune construction syntaxique supplémentaire n'est utilisée pour décrire les adaptations, des informations sur les contrôles sont également présentes. En effet, tout ce qui précède le `proceed` correspond à un bloc `before` et tout ce qui vient après correspond à un bloc `after`.

La première catégorie de contrôle, dont ceux proposés par les filtres de composition et Noah notamment, donne un pouvoir d'expression plus grand et a l'avantage de fixer la sémantique liée à une modification comportementale. Ce sont ces contrôles qui aident à mettre en œuvre la composition de plusieurs adaptations : la composition d'adaptations peut être exprimée en terme de composition de contrôles.

Expression des contrôles en ASL

ASL s'appuie sur la notion de contrôle pour exprimer l'application et la désapplication des schémas d'adaptation. Pour résumer, l'application d'une adaptation élémentaire de type **ajout de port** à un composant *c* revient à l'ajout du port fourni au rôle de *c* et la désapplication de cette adaptation à *c* revient au retrait de ce port fourni du rôle de *c* et correspond à une adaptation élémentaire de type **retrait de port**. Notons que ajouter un port ne signifie pas modifier l'état interne d'un composant. L'assemblage qui est formé ainsi que les composants qui en font partis constituent l'état du composant pour la gestion de la fonctionnalité liée à ce nouveau port. Notons également qu'un port ajouté ne peut pas être utilisé par d'autres adaptations élémentaires définies dans un même schéma.

De même, l'application d'une adaptation élémentaire de type **contrôle de port** permettant de modifier le comportement associé à un port à un composant *c* revient à composer le nouveau contrôle avec le contrôle existant sur le port d'adaptation correspondant du composant et la désapplication de cette adaptation à *c* revient à recomposer l'ensemble des contrôles exercés sur le port de *c* à l'exception du contrôle défini cette adaptation.

Les adaptations élémentaires du langage ASL ont la forme : <type> <coupe> '->' <contrôle> (voir la grammaire ASL en annexe A). *type* est le type de l'adaptation élémentaire (`modifyPort` pour une adaptation élémentaire de type **contrôle de port** et `newPort` pour une adaptation élémentaire de type **ajout de port**).

Expression des coupes *coupe* est de la forme ['!'] <identificateur de rôle> '.' <signature de port>. Notons que l'utilisation de '*' comme signature permet de dénoter n'importe quel port. Le '!' représente une autre forme de généricité et correspond au '*' excepté le port qui suit le '!'. Nous n'avons pas besoin du '*' au niveau de l'identificateur de rôle car un rôle ne dénote aucune classe en particulier. Ainsi, le schéma peut être appliqué à n'importe quel composant pouvant jouer ce rôle.

Nous n'avons pas besoin non plus des opérateurs logiques de conjonction et de disjonction. En effet, la conjonction (cf figure 6.4) qui permet de combiner plusieurs coupes devant toutes être « matchées », s'exprime par deux adaptations élémentaires définies dans un même schéma (la propriété d'atomicité garantit que les deux adaptations seront appliquées ou aucune ne le sera). De même, la disjonction (cf figure 6.5) qui permet de combiner plusieurs coupes dont au moins une doit être « matchée », s'exprime par deux adaptations élémentaires définies dans deux schémas d'adaptation différents qui pourront être appliqués indépendamment l'un de l'autre.

```
AdaptationPattern conjonction (A a, B b) {
   modifyPort a.m1() & b.m2() -> <contrôle>
}
```

```
AdaptationPattern equivalence_conjonction (A a, B b) {
   modifyPort a.m1() -> <contrôle> ,
   modifyPort b.m2() -> <contrôle>
}
```

FIG. 6.4 – Equivalence pour la conjonction

```
AdaptationPattern disjonction (A a, B b) {
   modifyPort a.m1() | a.m2() -> <contrôle>
}
```

```
AdaptationPattern equivalence_disjonction1 (A a, B b) {
   modifyPort a.m1() -> <contrôle>
}
AdaptationPattern equivalence_disjonction2 (A a, B b) {
   modifyPort a.m2() -> <contrôle>
}
```

FIG. 6.5 – Equivalence pour la disjonction

Expression des contrôles *contrôle* représente l'advice associé à l'adaptation élémentaire c'est à dire le code de traitement de l'adaptation. ASL est basé sur des opérateurs dont un sous-ensemble est commun avec ceux proposés par ISL [12]. La figure 6.6 présente la hiérarchie UML des contrôles ASL.

- l'opérateur « ; » (`SequentialControl`) désigne la séquence ordonnée,
- l'opérateur « / / » (`ShuffleControl`) désigne la séquence non ordonnée,
- l'opérateur « @ » (`WaitControl`) désigne l'attente. Un groupe de messages est contraint d'attendre la fin d'exécution d'un autre groupe de messages qui a été qualifié. La qualification, notée « [*i*] », permet d'identifier un groupe de messages à attendre,
- l'opérateur « _call » (`NotifyControl`) désigne le port correspondant au message à contrôler et correspond au proceed d'aspectJ [59],
- l'opérateur « delegate » (`DelegateControl`) est un opérateur désignant l'absence du _call. Il met en œuvre le remplacement du message d'origine qui n'est donc plus appelé directement,
- l'opérateur « exception » (`ExceptionControl`) désigne la levée d'une exception (le message d'origine n'est plus appelé),
- l'opérateur conditionnel (`ConditionnalControl`) est une combinaison conditionnée des opérateurs précédents. Notons que la condition ne doit pas comporter de port ayant un effet de bord et ne doit pas porter sur le typage d'un composant,
- l'opérateur « result » désigne le port dont le résultat est utilisé comme valeur de retour du port contrôlé ou ajouté.

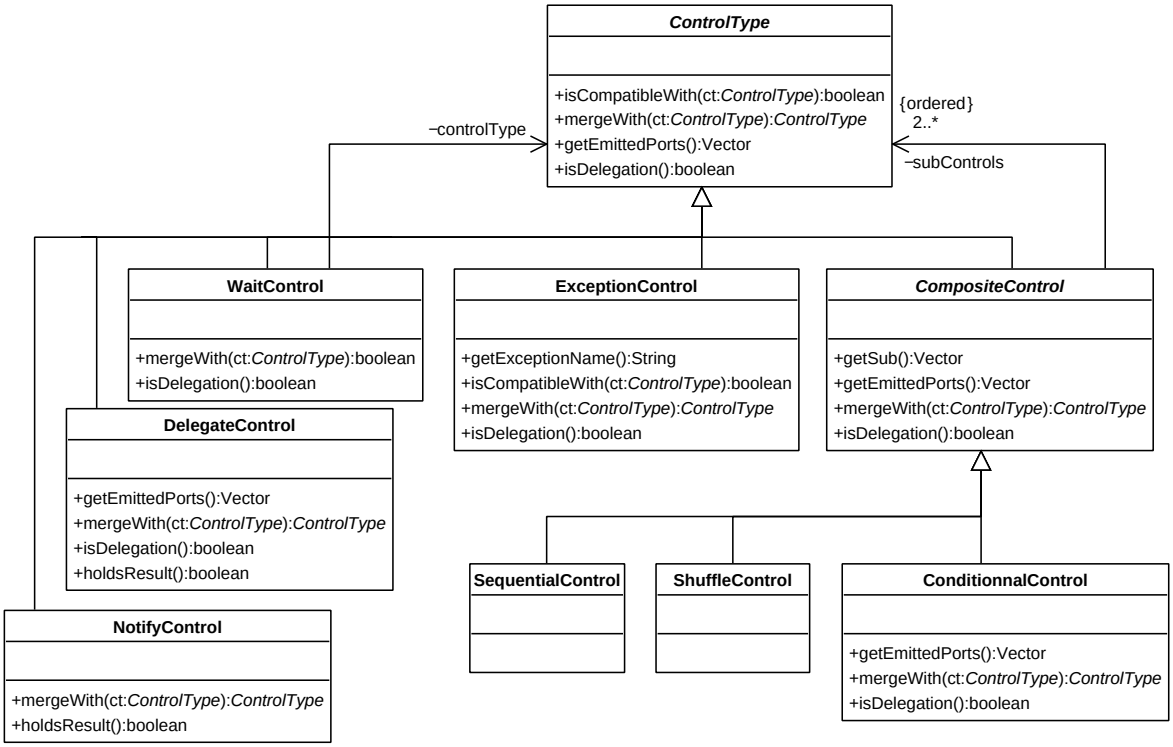

FIG. 6.6 – Hiérarchie UML des contrôles ASL

Notons que le langage ASL est une version « épurée » d'ISL qui ne prend en compte que les éléments permettant d'extrapoler une information de typage. Par exemple, dans la partie condition de l'opérateur conditionnel, ISL permet l'utilisation d'opérateurs comparatifs (=, <, >, ...) alors qu'ASL ne garde que la liste des messages faisant partis de la condition. L'opérateur `delegate` défini en ASL ne porte que sur un unique port alors que le même opérateur défini en ISL peut également porter sur un opérateur composite.

D'autre part, ASL inclut quelques constructions supplémentaires, toujours dans le but de recueillir un maximum de données en ce qui concerne le typage des adaptations. Par exemple, dans la définition de la coupe (messages à contrôler) le type de retour d'un message peut être déclaré (lorsqu'il existe et est utilisé dans l'adaptation) contrairement à ISL. De même l'opérateur `result` a été ajouté pour typer ce retour et permet d'exprimer la modification de la valeur de retour. ASL permet également d'exprimer la modification des paramètres d'appel du port contrôlé. Enfin, les ports émis peuvent être contrôlés comme les ports fournis ce qui permet de mettre en œuvre les adaptations concernant les messages sortant comme dans l'approche des filtres de composition [13] par exemple.

Pourquoi `ASLExtension` est un bon candidat pour être utilisé en tant que modèle d'adaptations

Le modèle `ASLExtension` est utilisé pour raffiner le modèle abstrait de Satin lorsque la projection vers une plate-forme n'est pas possible car celle-ci ne propose pas de modèle d'adaptations élémentaires conforme au modèle requis `AdaptationExtension` (elle ne fournit pas un équivalent aux opérations telles que `ElementaryAdaptation.isCompatibleWith`). La description picore correspondant au modèle `ASLExtension` est définie ci-après.

```
metamodel ASLExtension
    class ControlType
        operation match(Role r) : boolean
    ...
    class AdaptationRule
        operation isCompatibleWith(AdaptationRule ar) : boolean
        operation getProperties() : collection
        operation getType() : String

    class ControlRule extends AdaptationRule
        attribute control 1 ControlType
    class NewPortRule extends AdaptationRule
        attribute impl 1 ControlRule
```

La classe racine des adaptations élémentaires `AdaptationRule` fournit les opérations requises par le modèle abstrait pour introspecter une adaptation élémentaire et pour déterminer si deux adaptations élémentaires sont compatibles ou non. La classe racine des contrôles `ControlType` fournit l'opération permettant de déterminer si un contrôle concerne un rôle donné ou non. Le modèle `ASLExtension` est donc conforme au modèle requis d'adaptations.

Par ailleurs, un but du langage ASL est d'être suffisamment abstrait pour qu'il puisse exprimer les adaptations élémentaires dans un maximum de situations afin de pouvoir remplir son rôle de modèle d'adaptation par défaut à utiliser avec le modèle abstrait lorsque nécessaire. Le tableau 6.1 met en relation différents contrôles de certains modèles avec les contrôles ASL pour en montrer le pouvoir d'expression. Les modèles étudiés mettent en œuvre des adaptations élémentaires de la famille **composition comportementale** ou **évolution des interfaces**. Notons que les correspondances pour le modèle FAC ne sont pas présentées car ce sont les mêmes que pour JAC (les adaptations étant exprimées de la même manière).

	Contrôles du modèle	**Contrôles ASL correspondant**
CF	Dispatch = { a.*, b.* };	modifyPort a.* -> a._call(), modifyPort b.* -> b._call()
	Error = {true ∼> o.m};	modifyPort o.m -> exception
	Error = {true -> o.m};	modifyPort ! o.m -> exception
	Meta = {cond =>[a.*]o.m };	modifyPort a.* -> if cond then o.m(_call) else a._call endif
	Substitute = {[a.m] b.n) }	newPort a.m() -> b.n()
Noah	o.m(int i) -> a.m1() ; o._call ; b.m2()	modifyPort o.m(int i) -> a.m1() ; o._call(i) ; b.m2()
	o.m() -> a.m1(b.m2())	modifyPort o.m() -> C c = b.m2() ; a.m1(c)
	o.m() -> if (o.cond1() < o.cond2()) { a.m1() } else { b.m2() }	modifyPort o.m() -> if o.cond1() ; o.cond2() then a.m1() else b.m2() endif
AspectJ	before() : call(* O.m()) { a.m1() ; }	modifyPort o.m() -> a.m1() ; o._call()
	after() : call(* O.m()) { b.m2() ; }	modifyPort o.m() -> o._call() ; b.m2()
	around() : call(P O.m()) { Q q = a.m1() ; proceed() ; b.m2() ; return q ; }	modifyPort o.m() :P -> result a.m1() ; o._call() ; b.m2()
	public void O.m() { ...} où m est une méthode introduite dans l'aspect	newPort o.m() -> ...
JAC	a.m1() ; proceed() ; b.m2() où proceed = o.m()	modifyPort o.m -> a.m1() ; o._call() ; b.m2()

TAB. 6.1 – Mise en correspondance des contrôles ASL et ceux proposés par d'autres modèles

Positionnement de ASL vis-à-vis du problème de la composition comportementale

Dans le chapitre 3, nous avons vu que l'application d'un ensemble d'adaptations élémentaires visant à modifier le comportement d'une fonctionnalité d'un composant ne doit pas engendrer de conflits. Certaines adaptations élémentaires sont incompatibles et ne peuvent être appliquées conjointement à un composant. Il est donc important de pouvoir déterminer si la composition d'un ensemble d'adaptations élémentaires donné est possible.

Puisque ASL permet de contrôler les ports aussi bien à l'émission qu'à la réception, nous décomposons cette question de la manière suivante : comment prendre en compte la composition des contrôles d'un port `m` aussi bien quand il est fourni que quand il est émis.

- Si `m` est adapté seulement à la réception : la composition doit être validée en prenant en compte toutes les adaptations élémentaires concernant `m` dans le rôle du composant fournissant ce port.
- Si `m` est adapté seulement à l'appel (en tant que port émis) : pour chaque composant *c* qui utilise (appelle) `m`, la composition doit être validée en prenant en compte toutes les adaptations élémentaires concernant `m` dans le rôle de *c*.
- Si `m` est adapté à l'appel et à la réception, il faut composer les contrôles exercés sur le port `m` dans les cas mentionnés ci-dessus.

Notre approche consiste à n'accepter que les compositions d'adaptations élémentaires telles que quelque soit l'ordre d'application d'un ensemble d'adaptations élémentaires on obtienne toujours le même résultat. Nous pensons en effet que la commutativité et l'associativité sont des propriétés importantes surtout dans le cadre d'applications collaboratives où de multiples acteurs peuvent adapter l'application. Ces propriétés permettent également d'écarter certains conflits de composition.

Pour analyser le problème de la composition comportementale, examinons les différents cas qui peuvent se présenter. Le tableau 6.2 donne le résultat de la fusion pour le sous-ensemble d'opérateurs communs à ISL et ASL (la plupart des règles de composition décrites ici sont extraites de [12]). Seule la partie droite des adaptations élémentaires (correspondant au contrôle exercé) est inscrite dans le tableau. Le symbole `m` désigne le port contrôlé, `_call` représente l'exécution du port contrôlé, les termes en r_i désignent un opérateur de contrôle non terminal (séquence, conditionnelle, ...), C(x, y) indique qu'il faut recommencer la composition récursivement et le symbole $\times$ précise que la composition n'est pas possible.

Notons que la fusion ISL de deux exceptions est autorisée et a pour résultat la séquence non ordonnées des deux exceptions. Or, une seule exception peut être effectivement levée à l'exécution dans les plates-formes actuelles. Ainsi, suivant l'ordre d'application de deux adaptations élémentaires consistant à lever des exceptions différentes et suivant la façon dont l'opérateur « / / » est mis en œuvre, l'exception lancée n'est pas la même. La composition ASL se base sur les propriétés de commutativité et d'associativité pour détecter les conflits de type **exclusion mutuelle**). Pour cela, chaque couple de contrôles ne conservant pas ces deux propriétés est interdit. Deux adaptations élémentaires consistant à lever des exceptions différentes sont donc incompatibles en ASL.

Le fait que cette forme de composition s'apparente à une fusion plutôt qu'à un ordonnancement des adaptations élémentaires fait disparaître les conflits de type **exécution conditionnelle**. D'une part, les effets de bord dans les conditions ne sont pas autorisés. D'autre part, la fusion des opérateurs conditionnels permet de ne pas privilégier un contrôle conditionnel par rapport à un autre. Ainsi, lorsqu'une adaptation implique de composer deux contrôles de type conditionnels, l'adaptation n'est pas refusée mais nous informons l'adaptateur que cette composition peut poser problème si sa mise en œuvre dans la plate-forme est basée sur un ordonnancement des adaptations ou si les conditions ont un effet de bord.

	r3 ; m ; r4	r3 / / m / / r4	[2]r3 / / [3]m@2 / / r4@3	delegate	exception	if cond2 then r3 else r4 endif

r1 ; m ; r2	[0](r1 / / r3) / /
	[1]_call@0 / /
	(r2 / / r4)@1
		[0]r1 / / r3 / /
		[1]_call@0 / /
		r2@1 / / r4
			[0]r1 / / [2]r3 / /
			[3] _call@(0,2) / /
			r2@3 / / r4@3
				r1 ; delegate ; r2	exception	if cond2 then
						C((r1 ; m ; r2), r3)
						else
						C((r1 ; m ; r2), r4)
						endif

r1 / / m / / r2	r1 / / [0]r3 / /
	[1]_call@0 / /
	r2 / / r4@1
		r1 / / r3 / /
		_call / /
		r2 / / r4
			r1 / / [2]r3 / /
			[3]_call@2
			r2 / / r4@3
				r1 / / delegate / / r2	exception	if cond2 then
						C((r1 / / m / / r2), r3)
						else
						C((r1 / / m / / r2), r4)
						endif

[0]r1 / /
[1]m@0 / /
r2@1
	[0]r1 / / [2]r3 / /
	[1]_call@(0,2) / /
	r2@1 / / r4@1
		[0]r1 / / r3 / /
		[1]_call@0 / /
		r2@1 / / r4
			[0]r1 / / [2]r3 / /
			[1,3] _call@(0,2) / /
			r2@1 / / r4@3
				[0]r1 / /
				[1]delegate@0 / /
				r2@1
					exception	if cond2 then
						C(([0]r1 / / [1]m@0 / / r2@1), r3)
						else
						C(([0]r1 / / [1]m@0 / / r2@1), r4)
						endif

delegate	r3 ; delegate ; r4	r3 / / delegate / / r4	[2]r3 / / [3]delegate@2 / / r4@3	×	exception	if cond2 then C(delegate, r3) else C(delegate, r4) endif
exception	exception	exception	exception	exception	×	exception

if cond1
then r1
else r2
endif
	if cond1 then
	C(r1, (r3 ; m ; r4))
	else
	C(r2, (r3 ; m ; r4))
	endif
		if cond1 then
		C(r1, (r3 / / m / / r4))
		else
		C(r2, (r3 / / m / / r4))
		endif
			if cond1 then
			C(r1, ([2]r3 / / [3]m@2 / / r4@3))
			else
			C(r2, ([2]r3 / / [3]m@2 / / r4@3))
			endif
				if cond1 then
				C(r1, delegate)
				else
				C(r2, delegate)
				endif
					exception	if cond1 then
						if cond2 then C(r1, r3)
						else C(r1, r4) endif
						else
						if cond2 then C(r2, r3)
						else C(r2, r4) endif
						endif

TAB. 6.2 – Résultat de la composition des opérateurs communs à ASL et ISL

Tout comme la fusion ISL, l'opération de composition des opérateurs ASL est basée sur l'unification des paramètres des adaptations élémentaires. L'opérateur `_call` joue un rôle pivot et représente l'élément neutre pour l'opération de composition. Du fait de l'introduction de l'opérateur `result` dans ASL, l'unification ne peut se faire seulement sur `_call`. En effet, une adaptation élémentaire ne peut désigner qu'une seule valeur comme valeur de retour du port contrôlé. Ainsi, composer deux adaptations élémentaires implique de composer également leur valeur de retour. D'autre part, la redéfinition des paramètres d'appel a pour conséquence de différencier certaines occurrences de l'opérateur `_call`. C'est à dire que l'unification doit également se faire sur les paramètres du port contrôlé.

Le tableau 6.3 donne les nouvelles règles de composition pour la prise en compte de l'opérateur `result` et de la redéfinition des paramètres d'appel du port contrôlé. Dans le tableau, `m` désigne le port contrôlé et `i` représente le paramètre d'appel, les appels de port sont simplifiés : seul le nom du port est utilisé mais le receveur n'est pas spécifié (`o.m()` devient `m()`). Comme précédemment, le symbole × précise que la composition n'est pas possible.

a1	a2	C(a1, a2)
j= p() ; m(j)	m(i)	j= p() ; _call(j)
	r1 ; m(i) ; r2	[0]j= p() / / [1]r1 / / [2]_call(j)@(0,1) / / r2@2
	r1 / / m(i) / / r2	[0]j= p() / / r1 / / _call(j)@0 / / r2
	[0]r1 / / [1]m(i)@0 / / r2@1	[3]j= p() / / [0]r1 / / [1]_call(j)@(3,0) / / r2@1
	delegate	×
	exception	exception
	if cond then r1 else r2 endif	if cond then C(a1, r1) else C(a1, r2) endif
	k= o() ; m(k)	×
result m() ; k()	result delegate	result delegate ; k()
	m() ; result p()	[0]_call() / / result p()@0 / / k()@0
	r1 ; m() ; r2	[0]r1 / / [1]result _call()@0 / / r2@1 / / k()@1
	r1 / / m() / / r2	r1 / / [0]result _call() / / r2 / / k()@0
	[0]r1 / / [1]m()@0 / / r2@1	[0]r1 / / [1]result _call()@0 / / r2@1 / / k()@1
	delegate	result delegate ; k()
	exception	exception
	if cond then r1 else r2 endif	if cond then C(a1, r1) else C(a1, r2) endif
m() ; result k()	result delegate	×
	m() ; result p()	×
	r1 ; m() ; r2	[0]r1 / / [1]_call()@0 / / r2@1 / / result k()@1
	r1 / / m() / / r2	r1 / / [0]_call() / / r2 / / result k()@0
	[0]r1 / / [1]m()@0 / / r2@1	[0]r1 / / [1]_call()@0 / / r2@1 / / result k()@1
	delegate	delegate ; result k()
	exception	exception
	if cond then r1 else r2 endif	if cond then C(a1, r1) else C(a1, r2) endif
result delegate	result m() ; k()	result delegate ; k()
	m() ; result k()	×
	r1 ; m() ; r2	[0]r1 / / [1]result delegate@0 / / r2@1
	r1 / / m() / / r2	r1 / / [0]result delegate / / r2
	[0]r1 / / [1]m()@0 / / r2@1	[0]r1 / / [1]result delegate@0 / / r2@1
	delegate	×
	exception	exception
	if cond then r1 else r2 endif	if cond then C(a1, r1) else C(a1, r2) endif

TAB. 6.3 – Compléments sur la composition des opérateurs supplémentaires de ASL

Contrairement à ISL dont les contrôles ne concernent que la réception de messages, ASL permet aussi d'en contrôler l'émission. Ceci pose le problème de composition sous un nouvel angle. Lorsque des adaptations interviennent sur la même fonctionnalité mais que les moments de contrôle ne sont pas les mêmes, doit on opérer une composition globale ?

Le tableau 6.4 répertorie différentes situations qui peuvent se produire et donne le résultat de la composition globale dans le contexte d'un mode de communication synchrone. Par exemple, un contrôle à l'appel de `m` de la forme `r1 ; m ; r2` signifie que `r1` doit être exécutée avant l'envoi du message et que `r2` doit être exécutée quand la réponse à `m` est intervenue.

Dans le tableau, `m` désigne le port contrôlé, `_call` représente l'exécution du port contrôlé, les termes en r_i désignent un opérateur de contrôle non terminal et CG(x, y) indique qu'il faut recommencer la composition globale récursivement.

Contrôle à l'appel de `m`	Contrôle à la réception de `m`	Résultat de la composition double sur `m`
r1 ; m ; r2	r3 ; m ; r4	r1 ; r3 ; _call ; r4 ; r2
	r3 / / m / / r4	r1 ; (r3 / / _call / / r4) ; r2
	delegate	r1 ; delegate ; r2
	if cond then r3 else r4 endif	r1 ; if cond then CG(m ;r2, r3) else CG(m ;r2, r4) endif
r1 / / m / / r2	r3 ; m ; r4	r1 / / (r3 ; _call ; r4) / / r2
	r3 / / m / / r4	r1 / / r3 / / _call / / r4 / / r2
	delegate	r1 / / delegate / / r2
	if cond then r3 else r4 endif	r1 / / (if cond then r3 else r4 endif) / / r2
delegate	peu importe	delegate
peu importe	exception	×
exception	peu importe	exception
if cond then r1 else r2 endif	r3	if cond then CG(r1, r3) else CG(r2, r3) endif

TAB. 6.4 – Synthèse sur la composition globale

D'après le tableau 6.4, une gestion globale de la composition n'est pas nécessaire. Du point de vue du composant appelant `m`, tout se passe comme si le contrôle était exercé sur deux ports distincts. En effet, contrôler un appel à `m` sur un composant *c* revient à définir un port `m` sur soi-même qui appelle `m` sur *c* et à contrôler la réception de son propre `m`.

Cependant, dans le cas où un contrôle de type exception est exercé à la réception, il parait incorrect de vouloir effectuer un contrôle à l'émission de ce port. Dans ce cas, des actions sont entamées mais le déclenchement de l'exception les rend inutiles car empêche que le contrôle exercé s'exécute entièrement.

Quant au composant fournissant `m`, il n'est pas affecté par les contrôles exercés à l'émission sauf si lui-même contrôle ses propres appels à `m` et il se retrouve alors dans le cas du composant appelant qui ne pose pas de problème.

6.2.2 Recommandations pour la concrétisation d'un modèle d'adaptations

Une première exigence à remplir pour qu'un modèle d'adaptation soit choisi comme concrétisation du modèle requis est de fournir une implémentation pour chaque opération définie dans le modèle requis à l'exception du cas particulier de l'opération `ElementaryAdaptation.isCompatibleWith`. Si la propriété P_{4b} sur la cohérence de composition n'a pas besoin d'être vérifiée ou si le modèle d'adaptations ne prend pas en charge les adaptations de type **composition comportementale**, il n'est pas obligatoire d'implémenter cette opération.

Dans le cas contraire, le modèle d'adaptation doit vérifier que la composition n'engendre pas les conflits de type **exécution conditionnelle** et **exclusion mutuelle**. Mais il n'a pas à se préoccuper des conflits de type **anomalies de composition** et **récursivité accidentelle** car ces types de conflits sont déjà détecter au niveau du modèle abstrait à travers les propriétés de sûreté. En effet, les conflits de type **anomalies de composition** qui peuvent se produire avec les substitutions de type sont éliminés car la propriété P_3 sur la consistance des assemblages empêche qu'une adaptation puisse supprimer un port utilisé par une autre adaptation. Enfin, la propriété P_5 sur les cycles permet de détecter les conflits de type **récursivité accidentelle** qui sont engendrés par des coupes introduisant des éléments génériques.

Les modèles d'adaptations permettant de modifier le comportement d'un port aussi bien à l'émission qu'à la réception, tels que les filtres de composition [13] doivent en plus se poser la question de la nécessité de gérer une composition globales des adaptations.

Pour tout modèle d'adaptations élémentaires incluant un contrôle conditionnel tel que Noah [17] ou les filtres de composition [13] ou encore ASL (détaillé dans la section précédente), l'utilisation de ce type de contrôle est soumis à une restriction. En effet, pour que la déduction des rôles génériques à partir des adaptations élémentaires soit correcte, la partie *condition* d'un contrôle conditionnel ne doit pas porter sur le typage d'un composant. Pour comprendre cette contrainte, considérons le schéma `conditionnelle`, décrit en ASL, dont l'adaptation comportementale d'un port `m1` dépend du type du receveur. Cette adaptation ne dépend que du type du composant et les deux branches du 'if' ne seront jamais exécutées sur un même composant. Pourtant, la déduction du rôle `A` fait que le composant à qui on applique l'adaptation élémentaire et qui joue le rôle de `a` doit fournir les ports conformes à `m2` et `m3` alors que l'un de ces deux ports n'est jamais utilisé quelque soit le type du composant. Le composant ne joue pas le même « rôle » dans la branche 'then' et la branche 'else' et c'est son type qui détermine quelle branche appliquer ce qui n'a pas de sens.

```
AdaptationPattern conditionnelle (A a, B b) {
   modifyPort a.m1() -> if (a.instanceOf(...)) then a._call(); a.m2()
                        else a._call(); a.m3()
                        endif; c.m4()
}
```

De la même manière qu'il est recommandé, dans les programmes orientés objet, de ne pas faire de test sur le type des objets et de laisser faire le polymorphisme, nous recommandons de ne pas faire intervenir le type d'un composant dans la description d'une adaptation. Les schémas `equivalence_conditionnelle_then` et `equivalence_conditionnelle_else` permettent de réaliser une adaptation équivalente sans contraindre inutilement les composants. Suivant le type du composant on appliquera un des deux schémas.

```
AdaptationPattern equivalence_conditionnelle_then (A a, B b) {
   modifyPort a.m1() -> a._call(); a.m2(); b.m4()
}
AdaptationPattern equivalence_conditionnelle_else (A a, B b) {
   modifyPort a.m1() -> a._call(); a.m3(); b.m4()
}
```

6.2.3 Applicabilité par schémas versus par adaptations élémentaires

Les schémas d'adaptions sont utilisés comme unité d'application et de réutilisation des adaptations élémentaires. Cependant, il n'est pas toujours possible de garder l'unité d'un schéma. En effet, certains modèles d'adaptations ne permettent pas de garder suffisamment d'information pour que les schémas puissent être appliqués dans leur globalité.

Le fait que les schémas puissent être éclatés pose des problèmes de sûreté que nous allons détailler dans cette section. D'abord un exemple est utilisé pour illustrer le problème. Ensuite, nous établissons un certain nombre de définitions sur les schémas et montrons sur l'exemple qu'appliquer un schéma ne revient pas toujours à appliquer toutes les adaptations du schéma indépendamment les uns des autres. Pour des raisons de typage, il est en effet indispensable que les rôles attendus des composants (paramètres du schéma d'adaptation) soient déduits simultanément de toutes les adaptations élémentaires.

Exemple

Prenons l'exemple d'un schéma définissant deux adaptations élémentaires : l'une de type **ajout de port** et l'autre de type **contrôle de port**. Lorsqu'on applique le schéma dans sa totalité, on sait que le port **m** est un port ajouté. Le rôle `X` déduit ne contient pas **m** comme port fourni (les rôles ne sont déduits qu'à partir des ports utilisés dans une adaptation élémentaire soit des ports contrôlés par une adaptation élémentaire de type **contrôle de port**). Pour que le schéma soit applicable, le rôle du composant jouant le rôle `X` ne doit pas contenir **m** et doit être conforme à `X` (au sens du filtrage vu dans la section 6.1).

```
AdaptationPattern exemple (X x, Y y) {
   newPort x.m() -> ...                  (a1)
   modifyPort y.n() -> ...; x.m(); ...   (a2)
}
```

Par contre, lorsqu'on cherche à déterminer l'applicabilité des adaptations élémentaires une à une, tout dépend de l'ordre dans lequel on considère les adaptations élémentaires. L'ordre *a1* suivie de *a2* marche mais pas l'inverse car dans *a2* on ne sait pas que `m` est ajoutée donc le rôle `X` déduit de *a2* comporte `m` à ce moment là et :

- soit le rôle du composant ne va pas correspondre (s'il ne comporte pas `m` par exemple). Dans ce cas *a2* n'est pas applicable.
- soit il correspond (donc comporte `m`) et *a2* n'est pas applicable. Mais dans ce cas, *a1* n'est pas applicable puisque `m` est déjà présent dans le composant.

Formalisation du problème et définitions

Soit un schéma d'adaptation S comportant N adaptations élémentaires noté $S(a_1,\ldots,a_N)$. On note $\phi(S(a_1,\ldots,a_N),c_1,\ldots,c_M)$, la fonction d'application d'un schéma d'adaptation à un ensemble de M composants. On veut montrer que la formule suivante n'est pas vraie :

$\phi(S(a_1,\ldots,a_N),c_1,\ldots,c_M) \Leftrightarrow \phi(S(a_i),c_1,\ldots,c_M);\ldots;\phi(S(a_j),c_1,\ldots,c_M)$ $\forall$ séquence $\{i..j\}$ | $\{i..j\}$ est une permutation sur $\{1..N\}$ où le symbole ; représente le sequencement des actions.

Soit E_{AP}, l'ensemble des adaptations élémentaires de type **ajout de port** et E_{CP}, l'ensemble des adaptations élémentaires de type **contrôle de port**. Nous avons 3 cas :

1. $a_i \in E_{AP}\ \forall i | 1 \leq i \leq N$
2. $a_i \in E_{CP}\ \forall i | 1 \leq i \leq N$
3. $\exists a_i, a_j \mid 1 \leq i,j \leq N$ and $a_i \in E_{AP}$ and $a_j \in E_{CP}$

Les cas 1 et 2 ne posent aucun problème puisque toutes les adaptations élémentaires du schémas sont du même type : les adaptations élémentaires de type **ajout de port** sont orthogonales entres elles et la composition des adaptations élémentaires de type **contrôle de port** est associative et commutative donc l'ordre d'application n'a pas d'importance dans ces cas. Dans la prochaine section, nous analysons le cas 3 avec l'exemple du schéma `exemple` à l'aide d'un ensemble de définitions introduites ci-dessous.

Soit $\delta(\phi(S(a), c_1, \ldots, c_i, \ldots, c_M), i)$, la fonction qui renvoie le rôle attendu (rôle générique) du composant en position i pour l'application ϕ du schéma $S(a)$.

On note χ, la relation qui, étant donné un rôle générique et un composant, indique si c peut jouer le rôle r. Cette relation fait référence à l'opération `Component.canPlayRole` du modèle abstrait et utilise la relation de conformité au sens du filtrage vu dans la section 6.1 entre rôles.

On note ϵ, la fonction d'évolution qui étant donnée une adaptation élémentaire a et un composant c renvoie le composant avec un nouveau rôle associé. Soit $\rho(a)$, la position dans ϕ du composant auquel s'applique l'adaptation élémentaire a. Rappelons que $\Gamma(c)$ est l'ensemble des rôles associés au composant c (rôles concrets). $\forall i | 1 \leq i \leq M, \epsilon(a, c_i)$ renvoie c_i' tel :
- $\Gamma(c_i') \neq \Gamma(c_i)$ si $\rho(a) = i$ (a s'applique à ci dans $\phi(S(a), c_1, \ldots, c_i, \ldots, c_M)$)
- $\Gamma(c_i') = \Gamma(c_i)$ sinon

$\phi(S(a), c_1, \ldots, c_M)$ est **applicable** ssi :
$\forall i | 1 \leq i \leq M$ and $r = \delta(\phi(S(a), c_1, \ldots, c_M), i)$ and $\chi(r, c_i)$.

$\phi(S(a), c_1, \ldots, c_M)$ a pour **effet de bord** $\epsilon(a, c_i) \forall i | 1 \leq i \leq M$.

Démonstration par l'exemple

Dans cette section, nous reprenons l'exemple du schéma `exemple` comportant deux adaptations, *a1* et *a2*, et s'appliquant à deux composants. Soit $c_1 etc_2$, deux composants auxquels on cherche à appliquer le schéma. Dans la suite, supposons $\Gamma(c_1) = \{r_1\} | F(r_1) = \{o\}$ et $\Gamma(c_2) = \{r_2\} | F(r_2) = \{n\}$.

Soit $X_1 = \delta(\phi(exemple(a_1), c1, c2), 1)$, le rôle attendu de c_1 pour $exemple(a_1)$.
Soit $Y_1 = \delta(\phi(exemple(a_{1)}, c1, c2), 2)$, le rôle attendu de c_2 pour $exemple(a_1)$.
Soit $X_2 = \delta(\phi(exemple(a_2), c1, c2), 1)$, le rôle attendu de c_1 pour $exemple(a_2)$.
Soit $Y_2 = \delta(\phi(exemple(a_2), c1, c2), 2)$, le rôle attendu de c_2 pour $exemple(a_2)$.

En fonction des informations fournies par les adaptations élémentaires a_1 et a_2, on sait que : $I(X_1) = \{m\}$, $I(Y_1) = \{\}$, $m \in F(X_2)$, $I(X_2) = \{\}$, $n \in F(Y_2)$ et $I(Y_2) = \{\}$.

Dans l'exemple, $\rho(a_1) = 1$ car a_1 s'applique au premier paramètre du schéma, c'est à dire c_1 et $\rho(a_2) = 2$ car a_2 s'applique au second paramètre du schéma c'est à dire c_2.

Nous cherchons à prouver la commutativité de ϕ pour $N = 2$ et $M = 2$.
Si $\phi(exemple(a_1, a_2), c_1, c_2) \Leftrightarrow \phi(exemple(a_1), c_1, c_2); \phi(exemple(a_2), c_1, c_2)$
et si $\phi(exemple(a_1, a_2), c_1, c_2) \Leftrightarrow \phi(exemple(a_2), c_1, c_2); \phi(exemple(a_1), c_1, c_2)$
alors $\phi(exemple(a_1), c_1, c_2); \phi(exemple(a_2), c_1, c_2) \Leftrightarrow \phi(exemple(a_2), c_1, c_2); \phi(exemple(a_1), c_1, c_2)$

1) Considérons l'ordre d'application $\phi(exemple(a_1), c_1, c_2); \phi(exemple(a_2), c_1, c_2)$

$\phi(exemple(a_1), c_1, c_2)$ est applicable puisque on a $\chi(X_1, c_1)$ et $\chi(Y_1, c_2)$.
$\phi(exemple(a_1), c_1, c_2)$ a pour effet de bord :
– $c'_1 = \epsilon(a_1, c_1)$ tel que $\Gamma(c'_1) = \{r'_1\} | F(r'_1) = \{m, o\}$
– $c'_2 = \epsilon(a_1, c_2)$ tel que $\Gamma(c'_2) = \Gamma(c_2) = \{r_2\}$

$\phi(exemple(a_2), c_1, c_2)$ est applicable puisque on a $\chi(X_2, c'_1)$ et $\chi(Y_2, c_2)$.
$\phi(exemple(a_2), c_1, c_2)$ a pour effet de bord :
– $c''_1 = \epsilon(a_2, c_1)$ tel que $\Gamma(c''_1) = \Gamma(c'_1) = \{r'_1\}$
– $c''_2 = \epsilon(a_2, c_2)$ tel que $\Gamma(c''_2) = \{r'_2\} | F(r'_2) = \{n\}$

$\phi(exemple(a_1), c_1, c_2); \phi(exemple(a_2), c_1, c_2)$ fait évoluer les rôles de composants de la même façon que $\phi(exemple(a_1, a_2), c_1, c_2)$:
$\phi(exemple(a_1, a_2), c_1, c_2) \Leftrightarrow \phi(exemple(a_1), c_1, c_2); \phi(exemple(a_2), c_1, c_2)$.

2) Considérons l'ordre d'application $\phi(exemple(a_2), c_1, c_2); \phi(exemple(a_1), c_1, c_2)$

$\phi(exemple(a_2), c_1, c_2)$ n'est pas applicable puisque on a $\chi(Y_2, c_2)$ mais pas $\chi(X_2, c_1) : X_2$ contraint r_1 d'avoir m comme port fourni et ce n'est pas le cas.

Poursuivons tout de même en supposant $F(r_1) = \{m, o\}$, $\phi(exemple(a_2), c_1, c_2)$ devient applicable et a pour effet de bord :
– $c'_1 = \epsilon(a_2, c_1)$ tel que $\Gamma(c'_1) = \Gamma(c_1) = \{r_1\}$
– $c'_2 = \epsilon(a_2, c_2)$ tel que $\Gamma(c'_2) = \{r'_2\} | F(r'_2) = \{n\}$

$\phi(exemple(a_1), c_1, c_2)$ n'est pas applicable puisque on a $\chi(Y_1, c_2)$ mais pas $\chi(X_1, c_1) : X_1$ contraint r_1 de ne pas avoir m comme port fourni ce qui contredit l'hypothèse que nous avons pris pour que $\phi(exemple(a_2), c_1, c_2)$ soit applicable.
$\phi(exemple(a_1, a_2), c_1, c_2) \not\Leftrightarrow \phi(exemple(a_2), c_1, c_2); \phi(exemple(a_1), c_1, c_2)$.
$\phi(exemple(a_1), c_1, c_2); \phi(exemple(a_2), c_1, c_2) \not\Leftrightarrow \phi(exemple(a_2), c_1, c_2); \phi(exemple(a_1), c_1, c_2)$. CQFD

Cet exemple montre que lorsqu'il n'est pas possible d'appliquer les adaptations élémentaires d'un schéma dans la globalité, dans ce cas, les adaptations élémentaires de type **ajout de port** doivent être appliquées avant les adaptations élémentaires de type **contrôle de port** :

$\phi(exemple(a_1, \ldots, a_N), c_1, \ldots, c_M) \Leftrightarrow \phi(exemple(a_i), c_1, \ldots, c_M); \ldots; \phi(exemple(a_k), c_1, \ldots, c_M); \phi(exemple(a_{k+1}), c_1, \ldots, c_M); \ldots; \phi(exemple(a_j), c_1, \ldots, c_M)$ $\forall$ séquences $\{i..k\}, \{k+1..j\} \mid 1 \leq i, j, k, \leq N, a_i, \ldots, a_k \in E_{AP}, a_{k+1}, \ldots, a_j \in E_{CP}$.

Pour ne pas être confronté à ce problème d'ordonnancement, nous avons fait le choix pour le langage ASL d'interdire l'utilisation d'un port dans le même schéma où il a été défini.

6.2.4 Ancrage des rôles dans les adaptations élémentaires

Parfois certains rôles sont « ancrés » dans la définition des adaptations élémentaires, c'est à dire liés les uns aux autres. Cela arrive lorsque, par exemple, un rôle est utilisé en paramètre d'un port d'un autre rôle. Dans ce cas, lorsque le schéma d'adaptation est appliqué à une liste de composants, il est nécessaire de tenir compte des contraintes liant différents rôles pour ne pas violer les règles de typage de la plate-forme sous-jacente.

Exemple de rôles ancrés

Voici quelques adaptations élémentaires et les contraintes entre rôles qui en découlent. Pour simplifier la lecture des contraintes, on note *R.m*, le port de nom *m* d'un rôle *R*, *m.argsi*, le ième rôle de paramètre du port *m* et *m.return* le rôle de retour du port *m*. Il existe plusieurs cas de figure dont :

```
1) modifyPort a.m(B b) -> C c = d.m'(); a._call(c)          => D.m'.return ≤ A.m.args1
2) modifyPort a.m(): B -> C c = a._call(); d.m'(c)          => A.m.return ≤ D.m'.args1
3) modifyPort a.m(): B -> result c.m'()                     => C.m'.return ≤ A.m.return
4) modifyPort a.m(B b) -> c.m'(b)                           => A.m.args1 ≤ C.m'.args1
5) modifyPort a.m() -> B b = c.m'(); d.m''(b)               => C.m'.return ≤ D.m''.args1
```

Ainsi, l'adaptation élémentaire du schéma d'adaptation Synchronisation du chapitre 5 comporte deux contraintes liées aux rôles ancrés qui entrent dans le cas de figure 4.

```
adaptationPattern Synchronisation(AgendaSynchronisant a1, AgendaSynchronisé a2) {
 modifyPort a1.addRdv(Meeting m) -> if (a2.isFree(m)) then
                                       a1._call(m);
                                       a2.addRdv(m)
                                    else
                                       a1._call(m);
                                       a2.printError("non synchronisé")
                                    endif
}

AgendaSynchronisé.isFree.args1 ≤ AgendaSynchronisant.addRdv.args1
AgendaSynchronisé.addRdv.args1 ≤ AgendaSynchronisant.addRdv.args1
```

Rôles ancrés et applicabilité des adaptations élémentaires

Soit l'adaptation élémentaire : `modifyPort a.m(B b) -> c.m'(b)`

Les rôles génériques déduits sont A et C avec les contraintes suivantes :

$B = Any$[5]

$F(A) = \{m(Any)\}$

$F(C) = \{m'(Any)\}$

$A.m.args1 \leq C.m'.args1$

Supposons qu'on applique cette adaptation élémentaire à un composant c_1 en tant que A et à un composant c_2 en tant que C tels que :

$\Gamma(R_1) = \{c_1\} \mid \{m(int), m(String)\} \subseteq F(R_1)$

$\Gamma(R_2) = \{c_2\} \mid \{m'(String), m'(int)\} \subseteq F(R_2)$

Dans ce cas, l'adaptation élémentaire est applicable car :

$R_1 <\# A$

$R_2 <\# C$

$R_1.m.args1 \leq R_2.m'.args1$

[5] Aucune contrainte ne portant sur *B*, ce dernier correspond donc au rôle générique racine *Any*.

L'adaptation élémentaire s'applique comme suit :

```
c1.m(int b) -> c2.m'(b)
c1.m(String b) -> c2.m'(b)
```

Par contre, l'adaptation élémentaire n'est pas applicable si par exemple : $m'(String) \notin F(R_2)$

Dans ce cas, on a :

$R_1 <\# A$

$R_2 <\# C$

$R_1.m.args1 \not\leq R_2.m'.args1$ (la contrainte liée aux rôles ancrés n'est pas respectée)

On aurait pu appliquer l'adaptation élémentaire seulement à $c_1.m(String)$ mais cela irait à l'encontre de la propriété d'atomicité associée à la mise en œuvre des adaptations (cf. chapitre 5) qui veut que toutes les adaptations élémentaires soient appliquées simultanément ou que aucune ne le soit.

Rôles identiques et applicabilité des adaptations élémentaires

Dans notre approche, nous considérons que lorsque le même nom de rôle est utilisé à plusieurs endroits même si ce rôle correspond en fait au rôle générique racine *Any* alors lorsqu'on calcule l'applicabilité, on doit prendre en compte une autre contrainte de rôles ancrés : l'identité. Ainsi, si on a l'adaptation élémentaire : `modifyPort a.m() : B -> B b = c.m'()`

On doit vérifier, sur les composants auxquels on tente de l'appliquer, la contrainte :

$A.m.return = C.m'.return$

6.3 Discussion autour des rôles et des schémas d'adaptation

Certains points n'ont pas été modélisés à travers les rôles et les schémas d'adaptation. Dans cette section, nous justifions ce choix et discutons de l'impact de leur éventuelle modélisation sur le modèle abstrait et sur les modèles requis.

Types d'adaptations élémentaires. Si l'on prend en compte de nouveaux types d'adaptations élémentaires au niveau du modèle d'adaptation (que ce soit ASL ou un autre), on va être amené à étendre les rôles avec des informations supplémentaires dédiées à établir la sûreté des applications vis-à-vis des ces types adaptations. Nous montrons comment les deux relations de conformité sont étendues lorsqu'on prend en compte les adaptations liées aux composants nomades dans le chapitre 9.

Gestion des attributs Des modèles étudiés en 2, aucun ne permet l'ajout d'attributs à un composant. Nous avons donc privilégié la prise en compte des types d'adaptations élémentaires dont l'utilité est communément admise. Notre but étant avant tout de proposer un modèle de sûreté des adaptations aussi proche que possible des attentes des utilisateurs. Toutefois, la prise en compte des attributs au niveau des rôles peut être directe en considérant que ce sont des cas particuliers de ports fournis (avec des règles de conformité différentes).

Gestion de la visibilité. Pour l'instant, les adaptations que nous considérons ne concernent que les ports accessibles (au sens large du terme). Ainsi, seules les fonctionnalités publiques sont représentées au niveau des rôles. Cependant, nous pouvons nous poser les questions suivantes : Peut-on adapter une méthode privée ? Quand on a une adaptation telle que `modifyPort a.m() -> b.n() ; a.o()`, n ne peut sûrement pas être privée mais est ce que o peut être privée ? Doit on utiliser les visibilités pour autoriser ou non une adaptation ?

Catégories de ports. Bien que ce ne soit pas actuellement pris en compte au niveau du modèle abstrait, un port interdit peut représenter, au niveau d'un rôle concret, un port fourni ou émis du composant que l'on interdit d'adapter. De même, les ports émis peut représenter, au niveau d'un rôle abstrait, les fonctionnalités requises décrites au niveau des types de composant dans les plates-formes sous-jacentes.

Signature des ports. Actuellement, les ports sont identifiés par leur nom, un ensemble de paramètres et éventuellement un retour. Ce type de signature correspond au standard proposé par UML 2.0. Cependant, cela ne permet de représenter qu'un sous-ensemble des signatures de méthodes. La prise en compte de signatures comportant des paramètres in, out et in-out proposées entre autres par l'IDL OMG semble importante d'autant plus que ce type de signature englobe le précédent. L'extension du modèle abstrait implique alors essentiellement de reformuler la relation de conformité basée sur le filtrage (la relation de conformité basée sur la substituabilité ne change pas par contre puisqu'elle est déléguée à la plate-forme). D'autre part, le contrôle permettant de redéfinir le retour d'une adaptation élémentaire ainsi que sa composition avec les autres contrôles doivent être réétudiés dans le cas des modèles d'adaptation définissant ce type de contrôle puisqu'il peut y avoir plus qu'une valeur de retour. Dans la même lignée, on peut envisager de prendre en compte les signatures de ports avec valeur par défaut pour les paramètres.

Équivalence de ports. Si aucun port du rôle du composant ne correspond à un certain port d'un rôle du schéma, au lieu de dire que le composant ne peut pas jouer le rôle du schéma, on pourrait vouloir donner la possibilité d'établir une équivalence entre deux ports (par exemple, `save` et `sauvegarder`). Mais alors il faut se poser la question de savoir si cette équivalence est réutilisable et dans quel contexte : comment remplacer un composant pour lequel il existe des équivalences ?

Paramétrage des schémas d'adaptation. La plupart des langages orientés aspects introduisent une dépendance forte entre la définition d'un aspect et l'application sur laquelle l'aspect va être tissé en faisant référence à des noms de types, classes ou méthodes de l'application. Cette dépendance rend l'aspect non réutilisable contrairement à son objectif premier. Dans [60], il est démontré que l'utilisation des expressions régulières seule n'est pas suffisant pour rendre les aspects génériques car elles manquent d'expressivité et de précision. Du point de vue des types, les schémas d'adaptation sont suffisamment génériques par l'utilisation de la notion de rôle qui permet de ne pas faire référence aux types de l'application à adapter. Cependant, la définition d'un rôle s'appuie sur la définition de port qui fait référence aux fonctionnalités. A ce niveau de granularité, nous sommes alors amenés à faire de la correspondance de nom entre un port et une fonctionnalité issue de l'application. Pour rendre les schémas d'adaptation plus génériques du point de vue de l'utilisation des fonctionnalités et donc applicables dans un plus grand nombre de situations, nous devons être capable de rendre le nommage des ports utilisés dans un schéma indépendant du nommage utilisé dans l'application à adapter.

Par exemple, dans le schéma d'adaptation `exemple`, `M` ne désigne pas une fonctionnalité. Il s'agit en fait d'une variable permettant de décider de la fonctionnalité à utiliser seulement à l'application du schéma. `M` est en quelque sorte un lien symbolique qui n'est résolu qu'à l'application du schéma. Les rôles `A` et `B` sont alors déduits de l'adaptation élémentaire comme à l'ordinaire : `A`

comporte un port fourni `op1` et `B` un port fourni `M`. A l'application du schéma, il suffit de rajouter une étape supplémentaire à fin de recalculer les rôles vis-à-vis du contexte lié à l'application du schéma. Après quoi les vérifications des propriétés de sûreté sont effectuées comme s'il n'y avait pas de généricité. Dans l'application du schéma `exemple` aux composants *o1* et *o2* avec le contexte `M` = `op2`, *o1* doit pouvoir jouer le rôle de `A` (i.e. : fournir un port `op2`) et *o2* doit pouvoir jouer le rôle de `B` (i.e. : fournir un port `op2`). Ce contexte qui fixe les valeurs des fonctionnalités à utiliser doit faire parti de l'instance d'adaptation. En effet, deux applications du schéma `exemple` avec les deux mêmes composants mais deux valeurs différentes de `M` doivent être considérées comme deux adaptations différentes.

```
adaptationPattern exemple (A a, B b, M) {
a.op1() -> a._call(); b.M()
}
```

Notons que le paramétrage des schémas est une manière de palier au problème de l'équivalence des ports. Par exemple, au lieu de définir un schéma d'adaptation permettant de rendre des composants persistants en utilisant explicitement un port de nom « sauvegarder », il peut être utile d'utiliser un lien symbolique pour ce port. Ainsi, suivant le composant visé par l'adaptation, le schéma est appliqué en résolvant le lien symbolique par le nom « save », « sauvegarder », ...

« Ce que l'on conçoit bien s'énonce clairement et les mots pour le dire arrivent aisément »

Boileau

« Le sage n'affirme rien qu'il ne puisse prouver »

Proverbe latin

7

Formalisation et validation des propriétés de sûreté

POUVOIR aborder la problématique de la sûreté des adaptations d'applications de façon générale en exhibant les propriétés de sûreté que doivent satisfaire les adaptations, nous a paru être un axe de recherche important. Cependant la sûreté de fonctionnement étant un domaine sensible de l'informatique, il est nécessaire de mettre en œuvre des techniques destinées à fiabiliser notre solution, tant du point de vue de sa conception que de sa vérification. Surtout si celle-ci est utilisée pour contrôler l'évolution d'applications critiques en termes de risques humains (transports, équipements médicaux) ou économiques (communications, systèmes bancaires).

Les techniques de spécification et de vérification permettent une approche fiabilisée de ces problèmes. La spécification a pour objectif de proposer une méthodologie du développement visant à aider l'utilisateur à mieux maîtriser les différents aspects de l'application qu'il conçoit, en l'obligeant à suivre un certain nombre de règles. La vérification a pour but de réussir à prouver qu'un programme réalise bien ce qu'on a imaginé qu'il allait faire.

Dans le chapitre 5, nous avons vu que le modèle d'adaptations est spécifié en UML [91] alors que les propriétés de sûreté sont simplement énoncées en langage naturel. La section 7.1 précise comment formaliser les propriétés du Quoi et la section 7.2 comment valider cette expression formelle des propriétés par rapport aux propriétés elles-mêmes.

7.1 Formalisation

Spécifier un problème consiste à le décrire sans décider prématurément de la façon dont il sera résolu. Il existe plusieurs manières de spécifier un programme. Si l'on utilise un langage naturel alors la validation de programme n'est pas possible car cette activité repose sur l'utilisation d'une sémantique précise et non ambiguë. Concevant un modèle orienté objet avec UML [91], il nous a paru naturel d'utiliser OCL (Object Constraint Language) [120], le langage de contraintes intégré à UML, pour formaliser les propriétés du Quoi (P_0 à P_6). OCL est fondé sur la théorie des ensembles. Ce langage permet de spécifier des invariants de classes et des pré / postconditions d'opérations dans les modèles objets et n'a aucun effet de bord.

L'approche consiste à décrire des contraintes « comportementales » correspondant à un ensemble de préconditions OCL sur les opérations des classes du modèle Satin modélisant une forme d'adaptation. Ainsi, lorsqu'une telle opération est entamée, nous sommes sûr que l'adaptation en cours ne remet pas en cause la sûreté de fonctionnement de l'application concernée (dans le cas contraire, les préconditions de l'opération empêchent une adaptation invalide d'être effectuée). D'autre part, pour spécifier des restrictions sur l'architecture du modèle d'adaptations, nous avons également défini des contraintes « structurelles » [1] qui correspondent à un ensemble d'invariants de classes OCL. Le tableau 7.1 résume comment chaque propriété est assurée : par quelles contraintes comportementales, sur quelles opérations du modèle porte chaque contrainte et quelles opérations sont utilisées dans la définition de chaque contrainte.

Propriété de sûreté	**Contraintes associées**	**Opération sur laquelle porte chaque contrainte**	**Opérations utilisées pour chaque contrainte**
Conservation du contexte d'utilisation P_0	C_3	Component.replace	Role.isSubRoleOf
Adéquation des rôles vis-à-vis de l'implantation P_{1a}	C_{1a}	Template.createFromFacets	Role.isSubRoleOf
	C_{1b}	Template.createFromViewPoints	Role.isSubRoleOf
	C_{1c}	Template.createFromAdaptation	Port.filters
	C_4	AdaptationPattern.createFrom	ElementaryAdaptation.getType, ElementaryAdaptation.getProperties
Conservation des fonctionnalités de base P_{1b}	C_8	AdaptationPattern.instantiate	Component.canPlayRole
Garantie de visibilité P_2	C_{14}	AdaptationPattern.instantiate	Role.filters
Consistance des assemblages P_3	C_8	AdaptationPattern.instantiate	Component.canPlayRole
	C_{12}	AdaptationPattern.instantiate	AdaptationPattern.checkConstraint
	C_{15}	AdaptationInstance.remove	AdaptationInstance.containsDependencies
Uniformité comportementale P_{4a}	C_{2a}	Template.createFromFacets	-
	C_{2b}	Template.createFromViewPoints	-
Compatibilité des adaptations P_{4b}	C_5	AdaptationPattern.createFrom	ElementaryAdaptation.isCompatibleWith
	C_9	AdaptationPattern.instantiate	ElementaryAdaptation.isCompatibleWith
Points de non-déterminisme P_{4c}	C_6	AdaptationPattern.createFrom	AdaptationPattern.containsNDP
	C_{10}	AdaptationPattern.instantiate	AdaptationPattern.containsNDP
	C_{16}	AdaptationInstance.remove	AdaptationInstance.containsNDP
Cycles P_5	C_7	AdaptationPattern.createFrom	AdaptationPattern.containsCycleFrom
	C_{11}	AdaptationPattern.instantiate	AdaptationPattern.containsCycleFrom
	C_{17}	AdaptationInstance.remove	AdaptationInstance.containsCycleFrom
Rétro-activité des adaptations P_6	C_{13}	AdaptationPattern.instantiate	Port.filters

TAB. 7.1 – Contraintes comportementales associées à chaque propriété de sûreté

A titre d'exemple, nous détaillons dans ce chapitre les contraintes associées aux propriétés P_{1a} et P_{1b}, liées à la consomption des messages, et P_3 sur la consistance des assemblages. Les contraintes sont formulées en langage naturel puis leur traduction en OCL est explicitée.

Contraintes associées à la propriété P_{1a}

Les contraintes C_{1a}, C_{1b}, C_{1c} et C_4 sont associées à la propriété P_{1a} :

- C_{1a}, C_{1b} et C_{1c} : Toute fabrique à composant (`template`) connecte un ensemble de rôles génériques ou abstraits à une ou plusieurs implantations de façon à ce que chaque port des rôles soit associé à une fonctionnalité d'une implantation.
- C_4 : Toute adaptation élémentaire de type ajout de port utilise au moins un port.

[1] Par exemple, les composants, les schémas d'adaptation et les instances d'adaptation sont identifiées par leur nom et on ne peut pas avoir deux participants dans un schéma d'adaptation avec le même rôle.

Une contrainte est toujours associée à un élément de modèle : le contexte de la contrainte. Pour un invariant, le contexte spécifié est un nom de classe. Pour une pre-condition ou une post-condition, le contexte spécifié est un nom de classe suivi d'une signature d'opération. D'autre part, les contraintes sont nommées et le mot clé qui précède le nom de la contrainte indique s'il s'agit d'un invariant (`inv`), d'une pre-condition (`pre`) ou d'une post-condition (`post`). Par exemple, la contrainte C_{1a} définie ci-après est une pre-condition qui s'applique à l'opération `createFromFacettes` de la classe `Template`.

Le corps d'une contrainte correspond à un prédicat c'est à dire une expression qui renvoie un booléen. L'opérateur '.' permet de désigner des attributs ou des opérations d'objets et l'opérateur '->' pour désigner des attributs et opérations de collections (ordonnées ou non). Par exemple, le corps de la contrainte C_{1a} indique que tout rôle *r* appartenant à la collection *roles* doit pouvoir être substitué au rôle de l'implantation (à fin de s'assurer que tous les ports fournis par le rôle sont bien implémentés). De même, pour la contrainte C_{1b} mais cette fois plusieurs implantations sont à considérer et pour la contrainte C_{1c}, le rôle (générique) doit filtrer le rôle de l'implantation. L'opération `isSubRoleOf` (resp. `filters`) vérifie la conformité entre rôles au sens de la substituabilité (resp. au sens du filtrage) qui a été définie dans le chapitre 6. Notons que les ports des implantations ne correspondant à aucun port d'un rôle donné ne sont pas des ports fournis des composants instanciés par la fabrique et ne peuvent donc pas être adaptés ou utilisés dans une adaptation.

```
context Template::createFromFacets(roles : Collection(AbstractRole),
                                    implantation : Implantation) : Template
pre C1a :
    roles->forAll(r | implantation.implrole.isSubRoleOf(r))
```

```
context Template::createFromViewPoints(roles : Sequence(AbstractRole),
                                        implantations : Sequence(Implantation))
                                       : Template
pre C1b :
  (roles->size = implantations->size) and
  Sequence1..roles->size->forAll(index : Integer |
  implantations->at(index).implrole.isSubRoleOf(roles->at(index)))
```

```
context Template::createFromAdaptation(r : GenericRole, implantation : Implantation)
                                       : Template
pre C1c :
  r.filters(implantation.implrole)
```

Dans la contrainte C_4, l'opération getType indique le type de l'adaptation élémentaire et getType renvoie les ports utilisés par l'adaptation élémentaire.

```
context AdaptationPattern::createFrom(roles : Sequence(GenericRole),
                                      adaptations : Sequence(ElementaryAdaptation))
                                     : AdaptationPattern
pre C4:
  adaptations->forAll(a | a.getType() = 'add' implies a.getProperties()->notEmpty)
```

Contraintes associées à la propriété P_3

Les contraintes C_{12} et C_{15} sont associées à la propriété P_3 :

- C_{12} : Pour pouvoir appliquer un schéma d'adaptation à un ensemble de composants, il faut que chaque composant impliqué respecte les contraintes liées à l'ancrage des rôles (cf chapitre 6).
- C_{15} : Une instance d'adaptation créée à partir d'un schéma d'adaptation comportant une adaptation élémentaire de type ajout port ne peut être détruite tant qu'il existe d'autres instances d'adaptation créées à partir de schémas comportant une adaptation élémentaire impliquant le même composant et son port ajouté en question.

Dans la contrainte C_{12}, l'opération `checkConstraint` vérifie une contrainte de substituabilité entre deux rôles ancrés (`isSubRoleOf`). Tout composant dont le rôle du paramètre du schéma auquel il est associé est impliqué dans une contrainte doit la respecter. `self` désigne l'objet courant sur lequel la contrainte est évaluée, en l'occurrence une instance de la classe `AdaptationPattern`.

```
context AdaptationPattern::instantiate(components : Sequence(Component))
                                      : AdaptationInstance
pre C12:
  self.constraintList->forAll(c | c.checkConstraint(self.parameters, components))
```

Dans la contrainte C_{15}, l'opération `containsDependancies` vérifie si un port ajouté d'une instance d'adaptation est utilisé ou adapté dans une autre instance d'adaptation. Il est impossible de supprimer l'instance d'adaptation tant qu'il y a des dépendances.

```
context AdaptationInstance::remove()
pre C15:
  self.assignedadaptations->forAll(a | a.getType() = 'control' implies
                                  not self.containsDependencies(a))
```

Cas de la contrainte C_8 commune aux propriétés P_{1b} et P_3

La contrainte C_8 stipule que pour pouvoir appliquer un schéma d'adaptation à un ensemble de composants, il faut que chaque composant impliqué présente un rôle qui soit conforme au rôle du paramètre du schéma auquel il est associé. Dans la contrainte OCL décrite ci-dessous, l'opération `canPlayRole` se réfère à la relation de filtrage (`filters`) décrite dans le chapitre 6.

```
context AdaptationPattern::instantiate(components : Sequence(Component))
                                      : AdaptationInstance
pre C8 :
  Sequence{1..components->size}->forAll( index : Integer |
  components->at(index).canPlayRole(self.parameters->at(index)))
```

Cette contrainte sert à la fois à formaliser P_{1b} et P_3.

– Vis-à-vis de la propriété P_{1b}, la contrainte C_8 assure que le composant ne présente aucun des ports interdits du rôle auquel il est associé. En d'autres termes, le port à ajouter (le port interdit) d'une adaptation élémentaire de type ajout de port ne correspond à aucun port du composant auquel on applique l'adaptation élémentaire.
– Vis-à-vis de la propriété P_3, la contrainte C_8 assure que le composant comporte des ports fournis et émis conformes à ceux du rôle auquel il est associé.

Les autres contraintes en bref

Les définitions OCL des autres contraintes peuvent être consultées en annexe B. Nous dressons ici le listing de ces contraintes regroupées par propriétés et formulées en langage naturel. La figure 7.1 indique sur quelle opération du modèle abstrait porte chaque contrainte comportementale.

- Contraintes associées à la propriété P_0 sur la conservation du contexte d'utilisation :
  - C_3 : Pour pouvoir remplacer un composant c par un composant c' dans un assemblage, le rôle de c' doit être sous-rôle du rôle de c.
- Contraintes associées à la propriété P_2 sur la garantie de visibilité :
  - C_{14} : Cette contrainte permet d'assurer que les composants offrent un seul point d'accès pour toutes les fonctionnalités utilisées par les schémas d'adaptations. Pour cela, pour chaque composant c auquel on applique un schéma d'adaptation, il doit exister un rôle r associé à c tel que le rôle du schéma que c est censé jouer filtre r.
- Contraintes associées à la propriété P_{4a} sur l'uniformité comportementale :
  - C_{2a} **et** C_{2b} : Ces contraintes permettent d'assurer que les composants n'offrent qu'une seule version pour chaque fonctionnalité. Pour cela, les fabriques à composants ne doivent créer les composants qu'à partir de rôles dont l'intersection deux à deux sur l'ensemble des ports fournis est vide.
- Contraintes associées à la propriété P_{4b} sur la compatibilité des adaptations élémentaires :
  - C_5 : Pour tout couple d'adaptations élémentaires appartenant à un même schéma d'adaptation dont l'intersection des coupes est non vide, les contrôles des adaptations élémentaires doivent être compatibles.
  - C_9 : Pour toute adaptation élémentaire *ae* d'un schéma d'adaptation dont l'intersection de coupes avec au moins un port d'adaptation du composant auquel on applique *ae* est non vide, les contrôles de l'adaptation élémentaire et de chaque port d'adaptation concerné doivent être compatibles.
- Contraintes associées à la propriété P_{4c} sur les points de non-déterminisme :
  - C_6 : Pour toute adaptation élémentaire comportant des ports utilisés de manière non ordonnée, l'intersection de l'ensemble des ports atteignables depuis chaque port d'un ensemble de ports non ordonnés doit être vide. Les ensembles de ports non ordonnés d'une adaptation élémentaire sont donnés par l'opération `ElementaryAdaptation.withoutOrderingFor` du modèle requis d'adaptations.
  - C_{10} : Cette contrainte est la même que la précédente sauf que cette fois on prend en compte, dans le calcul des ports atteignables, les ports utilisés (sans contrainte d'ordre) par le composant à qui on applique l'adaptation élémentaire.
  - C_{16} : Cette contrainte est la même que la précédente sauf que cette fois on ne prend en compte, dans le calcul des ports atteignables, que les ports des composants.
- Contraintes associées à la propriété P_5 sur les cycles :
  - C_7 : Pour une adaptation élémentaire de type **contrôle de port** visant à contrôler un port p, l'ensemble des ports atteignables depuis p ne doit pas inclure p.
  - C_{11} : Cette contrainte est la même que la précédente sauf que cette fois on prend en compte, dans le calcul des ports atteignables, les ports utilisés par les composants en plus des ports utiliser dans l'adaptation élémentaire.
  - C_{17} : Cette contrainte est la même que la précédente sauf que cette fois on ne prend en compte, dans le calcul des ports atteignables, que les ports utilisés par les composants.

- Contraintes associées à la propriété P_6 sur la rétroactivité des adaptations :
  - C_{13} : Soit p un port à ajouter à un composant c. S'il existe au moins une instance d'adaptation dont c est un participant et qui comporte une adaptation élémentaire ae de type **contrôle de port** appliquée à c et si p correspond à la coupe de ae alors le remplacement de c par c' (où c' représente le composant c auquel on a ajouté le port p) doit être applicable.

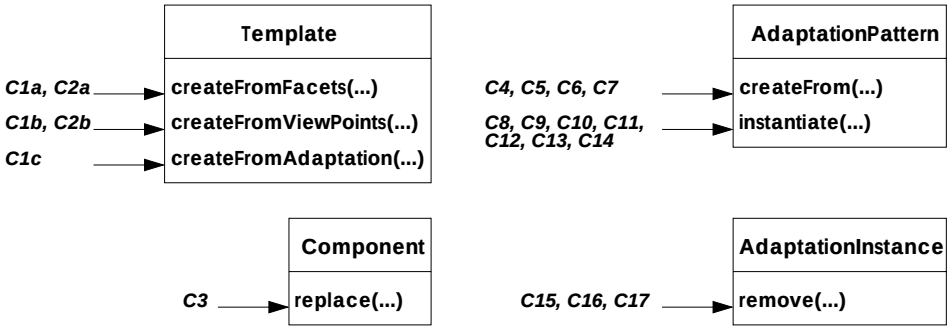

FIG. 7.1 – Contraintes comportementals associées aux opérations définies dans le modèle abstrait

Pour conclure, notons que le but de cette formalisation des propriétés par description de contrats d'adaptation consiste à simplifier la validation des propriétés de sûreté en la réduisant à une validation des contraintes OCL. La section suivante adresse ce problème.

7.2 Techniques de validation

A chaque propriété est associé un ensemble de contraintes OCL. Cependant, on ne peut pas simplement supposer ces équivalences correctes et attendre la phase de réalisation pour les constater. Il serait en effet difficile de détecter des bugs liés aux contraintes car des erreurs d'implémentation peuvent parasiter. Valider le modèle UML et les contraintes OCL consiste donc à s'assurer que les contraintes couvrent exactement la propriété de sûreté à laquelle elles sont associées (**complétude**) et que les contraintes d'une propriété n'interfèrent avec celles d'une autre propriété (**consistance**).

Il existe principalement deux techniques de validation : la simulation et la preuve. Nous avons expérimenté ces deux techniques. Nous avons commencé par la méthode la plus légère, la simulation, pour obtenir une première validation rapidement. Après en avoir étudier le potentiel et les limites face à notre problématique, nous avons cherché à améliorer les résultats obtenus en affinant la validation par la preuve sur les points qui paraissaient les plus sensibles. La section 7.2.1 présente nos résultats concernant la validation basée sur la simulation et la section 7.2.2 décrit notre démarche en ce qui concerne la preuve.

7.2.1 Simulation

Objectif de la simulation

Les techniques de simulation consistent comme leur nom l'indique à simuler le déroulement d'un programme avec un ensemble de données en entrée et à comparer les résultats obtenus avec les résultats attendus. Si les résultats divergent, il faut localiser et corriger l'erreur par rétro conception. Le point faible de cette technique est dû à la façon dont sont comparés les résultats. Généralement, la comparaison est faite par un humain avec toutes les sources d'erreurs supplémentaires que cela implique. De plus, il est difficile de savoir quand arrêter la simulation.

Outils utilisés

Notre objectif était de rester dans un standard de la MDA pour bénéficier de ses outils et de ne pas avoir à jongler avec plusieurs formalismes pour éviter des erreurs supplémentaires liées à la nécessité de synchroniser différentes versions d'une même spécification. Nous avons donc écarté le framework Objecteering [111] qui permet de tester dynamiquement l'intégrité des contraintes mais qui impose de définir les contraintes dans le langage propriétaire spécifique J. USE (UML-based Specification Environment) [102] est apparu être l'outil adéquat puisqu'il permet de simuler des contraintes écrites en OCL standard.

USE permet d'évaluer les contraintes sans avoir à implémenter le modèle complètement : seules les opérations utilisées dans les contraintes (et qui n'ont aucun effet de bord) sont décrites elles aussi en OCL. Mise à part la possibilité de valider la syntaxe et le typage des contraintes, la simulation consiste à construire explicitement des « configurations » (snapshots) représentant certains états d'un système. Une configuration est un ensemble d'objets avec des valeurs d'attributs déterminées et des associations entre objets. Pour une configuration donnée, si au moins une des contraintes testées est évaluée à « faux » ou a un résultat indéfini, l'état correspondant est considéré illégal et la configuration est rejetée. Notons que pour simuler le modèle Satin, il est nécessaire de choisir une concrétisation particulière des modèles requis. Le système de type de Java a été choisi comme concrétisation du modèle de contrat, les expressions régulières pour la concrétisation du modèle de coupe et le langage ASL (cf chapitre 6) comme concrétisation du modèle d'adaptations élémentaires.

Méthodologie : Recherche de contres exemples

Deux étapes sont nécessaires pour prouver qu'un ensemble de contraintes n'est pas consistant par rapport à une propriété donnée.

- Il faut vérifier que les contraintes ne soient pas trop faibles[2]. Pour cela, il faut rechercher des contre-exemples, c'est à dire, des configurations rejetées par une propriété P mais qui sont acceptées par la simulation des contraintes associées à P.
- Il faut vérifier que les contraintes ne soient pas trop fortes[3]. Cette fois encore, il faut rechercher des contre-exemples, c'est à dire des configurations acceptées par une propriété P mais qui sont rejetées par la simulation des contraintes associées à P.

Pour ne pas avoir à créer d'objets explicitement et valider rapidement un grand nombre de configurations, nous utilisons le langage déclaratif ASSL (A Snapshot Sequence Language) [44] au dessus de USE pour générer les configurations aléatoirement à partir de données spécifiées par l'utilisateur. Ceci permet d'explorer de manière systématique un sous-espace d'états. De plus, ASSL permet le backtracking, c'est à dire que si les objets créés ne vérifient pas une certaine contrainte, on cherche une autre configuration possible à partir des données de l'utilisateur. Toutes les combinaisons sont testées jusqu'à ce qu'un état valide soit trouvé. Ainsi, les contre-exemples peuvent être trouvés plus rapidement.

Néanmoins, comme ASSL interdit les effets de bord, seulement les invariants peuvent être testés. Nous avons donc dû transformer les préconditions correspondant aux contraintes comportementales en invariants. Par exemple, les deux préconditions correspondant aux contraintes C_5 et C_7 portent sur l'opération `createFrom` de la classe `AdaptationPattern` donc sur les paramètres `roles` et `rules`. Il est très facile de transformer ces deux préconditions en invariants car les paramètres `roles` et`rules` appartiennent à l'état des objets de la classe `AdaptationPattern`. Il faut juste ajouter une condition supplémentaire pour indiquer que les associations de `AdaptationPattern` avec ses `roles` et ses `rules` doivent être effectives. Sinon, la première configuration construite

[2] Aucun état indésirable ne doit être accepté par les contraintes.
[3] Aucun état valide ne doit être rejeté par les contraintes.

(toujours la même : celle où aucune association entre objets n'est effective) est toujours choisie et l'exploration des espaces d'états se terminera. Dans notre cas, la transformation des préconditions en invariants n'a pas posé pas de problème, la sémantique des contraintes n'a pas été perdue au cours de la transformation. De plus, cette transformation ne sert qu'à l'étape de validation (en pratique, à la projection, ce sont bien des préconditions qui sont utilisées).

Illustration

Par exemple, une fabrique à composants (`template`) peut être créée à partir d'un ensemble de rôles et d'une implantation (`createFromFacets`). La contrainte C_{1a} liée à cette opération, assure qu'une fabrique ne puisse pas être créée à partir d'une implantation et d'un ensemble de rôles dont certains ports ne correspondraient à aucun port de l'implantation. La contrainte C_{1a} correspond donc à la préservation de la propriété de sûreté P_{1a} sur l'adéquation des rôles vis-à-vis de l'implantation. Pour tester la complétude de cette contrainte avec P_{1a}, on utilise une procédure, `testCreateTemplate`, qui permet de créer des fabriques à partir d'un ensemble de rôles et d'une implantation (voir le code de la procédure en annexe C). `testCreateTemplate` prend quatre paramètres :

1. un ensemble de ports potentiels,
2. le nombre de rôles,
3. le nombre de ports par rôle,
4. le nombre de ports dans l'implantation.

Nous avons implémenté cette procédure de façon à ce que les ports des rôles soient fixés au départ en les choisissant aléatoirement dans l'ensemble des ports potentiels donné par l'utilisateur. Les ports de l'implantation sont eux aussi choisis aléatoirement dans l'ensemble des ports potentiels mais c'est sur cet élément que l'on va faire du backtracking : on teste différentes combinaison de ports dans l'implantation jusqu'à ce que l'on trouve une configuration acceptée par la contrainte C_{1a}.

La procédure `testCreateTemplate` a été lancée avec comme ensemble des ports potentiels 'm1','m2','m3','m4','m5'}, deux rôles comportant deux ports chacun et une implantation comportant 3 ports. Dans ce cas, le nombre maximal de configurations pour l'implantation correspond au choix de 3 éléments parmi 5 possibles (5 possibilités pour le premier port de l'implantation, 4 pour le second et 3 pour le dernier). Il y a donc en tout 60 possibilités à tester au plus à chaque lancement de la procédure. Bien sûr si on change le nombre de ports par rôle ou le nombre de ports dans l'implantation, on va réduire ou augmenter le nombre de configurations à évaluer. La figure 7.2 présente le résultat d'un lancement de la procédure `testCreateTemplate` avec les paramètres indiqués précédemment.

Cet exemple met en évidence un désavantage de la simulation. Dans ce cas précis, 24 configurations ont été générées et rejetées vis-à-vis de la contrainte testée. Or, il n'est pas sûr qu'il était légitime que ces 24 premières configurations soient rejetées. Pour s'en convaincre, il faut les examiner une à une, ce qui pose problème quand on a un grand nombre de configurations rejetées par l'outil. Plutôt que de chercher des configurations qui doivent être acceptées, on peut vérifier que des configurations incorrectes sont bien rejetées. Par exemple, si on lance la procédure en changeant le nombre de ports par rôle et qu'on le passe de 2 à 4, on est certain que toutes les configurations générées sont forcément incorrectes[4] et doivent être rejetées si la contrainte est bien écrite. La contrainte C_{1a} a été évaluée par ce biais avec plusieurs valeurs de ports et aucune des configurations ainsi créées n'a été acceptée par la contrainte comme nous l'attendions. La figure 7.3 présente le résultat de cette simulation.

[4]Puisque l'implantation comporte seulement 3 ports, il y a forcément au moins un port dans chaque rôle qui ne correspond à aucun port de l'implantation.

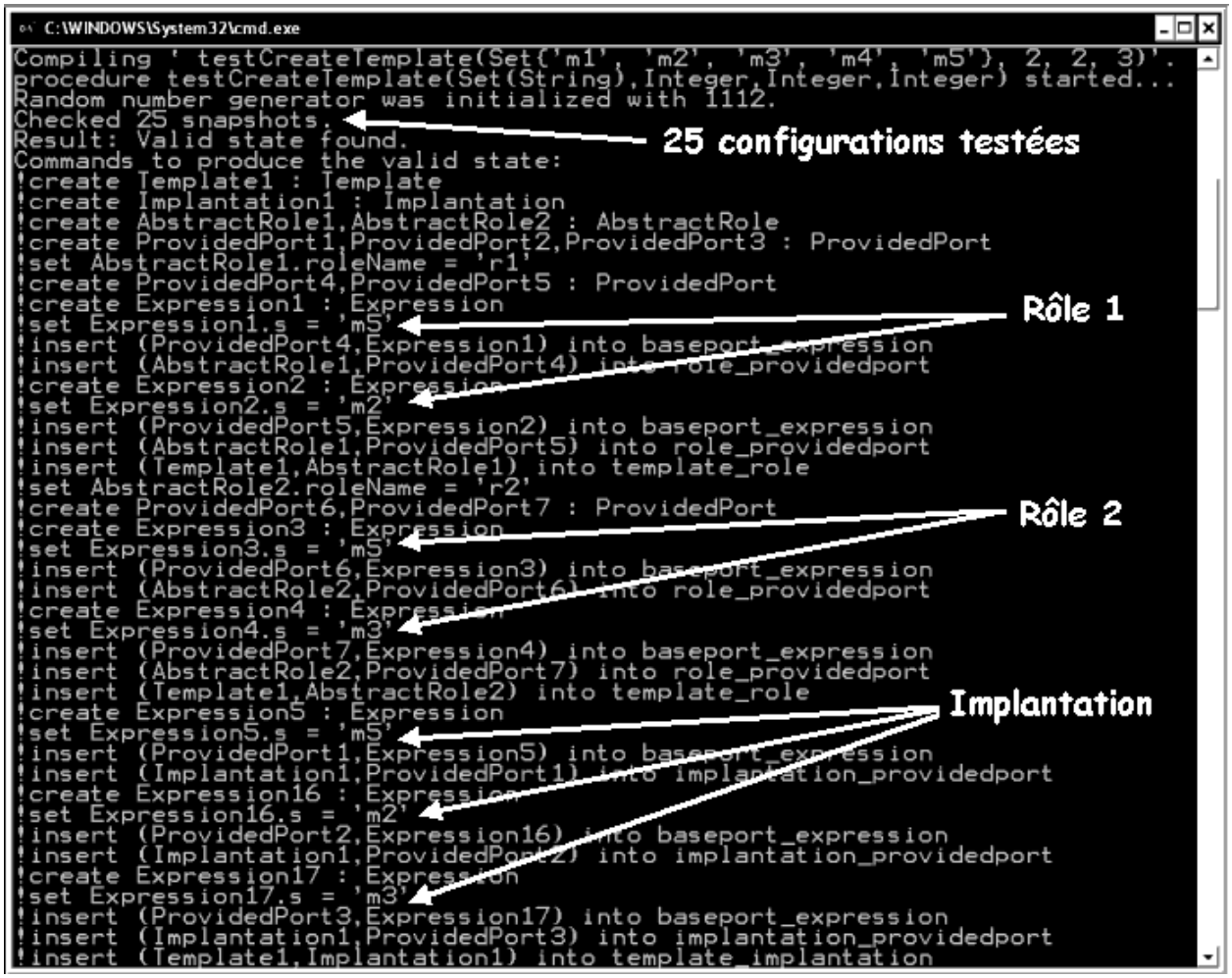

FIG. 7.2 – Résultat d'une évaluation de `testCreateTemplate`

FIG. 7.3 – Résultat d'une autre évaluation de `testCreateTemplate`

Evaluation

En analysant les états acceptés et les états rejetés, nous avons pu améliorer notre spécification et ainsi assurer rapidement un certain degré de confiance dans la spécification à un moindre coût puisqu'il n'est pas nécessaire d'implémenter le modèle et les contraintes. La simulation est intéressante pour trouver les configurations incorrectes mais reste fastidieuse pour trouver les configurations correctes. En effet, pour déterminer la non-consistance d'un ensemble de contraintes vis-à-vis d'une propriété, il suffit de trouver au moins un contre-exemple. Par contre, la consistance n'est garantie que vis-à-vis des états analysés et l'absence de contre exemple ne prouve rien.

Pour améliorer la validation des configurations correctes, une idée consiste à traduire les propriétés elles-mêmes en invariants OCL puis à tester à la fois les configurations par rapport à cet invariant et la *négation* des contraintes qui s'y rattachent. S'il y a complétude, toutes les configurations doivent être rejetées. Dans notre cas, cette technique de validation n'est pas utilisable car les propriétés de sûreté et les contraintes ne portent pas sur les mêmes objets : les propriétés sont toutes rattachées sur les composants ce qui est normal puisque c'est bien le fonctionnement des composants dont on veut préserver la sûreté alors que les contraintes portent sur les opérations du modèle qui permettent de créer, détruire et adapter les composants.

La simulation avec USE est utile pour détecter rapidement des erreurs dans la formalisation des contraintes en OCL. Cette technique de validation permet de ne pas avoir recours au prototypage pour faire du test, plus long à mettre en place, même si finalement les deux solutions conduisent aux mêmes possibilités en terme de correction. La simulation est donc une approche très pragmatique. Cependant, simuler dans un certain nombre de cas extrêmes pour s'assurer qu'il réagit conformément à ce qu'on attendait n'est pas suffisant pour réellement prouver de manière exhaustive qu'une spécification est correcte. L'exactitude de la spécification est en effet garantie seulement vis-à-vis des états analysés. La prochaine section montre comment nous avons complété la validation de la modélisation en utilisant la technique de preuves par théorèmes.

7.2.2 Preuves par théorèmes

Objectif de la preuve

Les techniques de preuves sont basées sur des démonstrations mathématiques. Un certain nombre d'axiomes et de règles d'inférences permettent d'aboutir à des conclusions (ce que l'on veut prouver). L'utilisation des techniques de preuves est difficile du fait que les démonstrateurs de théorèmes sont rarement complètement automatisables. En effet, l'utilisateur est souvent obligé de guider la preuve en interagissant avec le démonstrateur de théorèmes. A ce prix, il est possible d'établir formellement la preuve de l'exactitude d'une spécification.

Techniques de preuves et outils utilisés

Nous avons étudié plusieurs outils permettant de faire des preuves. Comme nous avons écarté l'outil de simulation Objecteering [111] à cause de l'absence de prise en charge de contraintes standard, nous avons mis de coté Jack [22], un outil de preuves, parce que les contraintes sont décrites en JML [65] et ne sont dédiées qu'à être utilisées au dessus de Java. Key [4], un autre outil de preuves basé sur de l'OCL standard a également attiré notre attention. Malheureusement, les preuves qu'il permet de faire ne correspondent pas à nos besoins. N'incluant pas de cas d'héritage de contraintes, notre modèle ne nécessite donc pas d'être soumis à la vérification du sous-typage structurel et comportemental proposé par l'outil. D'autre part, la préservation d'invariants de classe n'est pas utilisable car, comme nous l'avons dit dans la section précédente, les contraintes ne portent pas sur les mêmes classes que les invariants (propriétés de sûreté) qu'elles doivent préserver. Nous nous sommes finalement tourné vers B dont l'avantage est la possibilité de modifier ce découpage.

La méthode B [1] est une méthode de spécification et de vérification. Cette méthode est basée sur la théorie des ensembles. Elle est non seulement une méthode de spécification puisqu'elle propose une démarche de développement basée sur la notion de raffinement (un modèle initial abstrait est progressivement enrichi jusqu'à obtenir une spécification directement implantable), mais elle est aussi une méthode de vérification car elle permet de spécifier des propriétés invariantes devant être respectées à tout moment par le système.

La notion de machine abstraite est le concept de base de la méthode B. Elle contient des ensembles constants et variables, un invariant (conjonction de prédicats) et des opérations permettant d'accéder et de modifier les ensembles variables et dont la précondition (conjonction de prédicats) doit préserver l'invariant de machine. Les propriétés invariantes qu'il est possible de vérifier sur ces données sont exprimées à l'aide de la logique du premier ordre. La modification des données est définie à l'aide de substitutions généralisées.

Les obligations de preuve (OP) sont des formules logiques, il en existe de deux types. L'obligation de preuve pour l'initialisation est de la forme $[B]I$ où B représente un ensemble d'affectations des ensembles variables (spécifiées dans la partie *initialisations* de la machine B). L'OP pour l'initialisation permet de vérifier que les initialisations de ces ensembles permettent bien d'établir l'invariant I. Une obligation de preuve pour une opération de précondition Q et d'action V est de la forme $I\&Q => [V]I$ où $[V]I$ est la substitution représentant la formule I dans laquelle toutes les occurrences libres d'ensembles affectés dans V sont remplacées par leur nouvelle valeur dans V. Les OPs pour une opération permettent de vérifier qu'un appel légitime (vérifiant Q) sur un état correct (I) rend bien un état correct ($[V]I$).

Comme une machine peut correspondre à une classe aussi bien qu'à un système plus complexe, l'utilisation de ce formalisme permet de modifier le découpage des classes. Utiliser une machine pour modéliser plusieurs classes est un avantage pour nous. En effet, si l'on associait une machine à chaque classe du modèle Satin, la préservation de l'invariant de machine ne nous permettrait pas de vérifier la consistance et la complétude des contraintes car les contraintes ne portent pas sur les mêmes classes que les invariants (propriétés de sûreté) qu'elles doivent préserver. A l'inverse, associer plusieurs classes à une même machine permet de vérifier que les appels légitimes des opérations (respectant la précondition des opération c'est à dire toutes nos contraintes comportementales) d'une machine préservent l'invariant de machine (la conjonction de toutes nos propriétés de sûreté) via la génération d'obligations de preuve qu'il faut prouver.

Méthodologie concernant la transformation UML/OCL vers B

Notre expérimentation en ce qui concerne les transformations a permis de tirer quelques leçons. La spécification B qui prend en compte tous les éléments de la spécification UML / OCL pour prouver les propriétés P_{1a} sur l'adéquation des rôles vis-à-vis de l'implantation et P_3 sur la consistance des assemblages peut être consultée en annexe D.

Le point de départ est de « repenser » la modélisation, non pas en terme d'entités mais en terme d'ensembles. Ainsi, une classe correspond à un ensemble et une association entre deux classes correspond à une relation qui est également un ensemble. Un ensemble constant (*sets*) comporte alors toutes les instances possibles d'une classe et un ensemble variable (*variables*) représente les instances présentes dans le système à un instant t. Le même principe s'applique aux relations. Par exemple, dans notre spécification, l'ensemble constant $COMPONENTS$ représente les instances qu'il est possible de créer (ne peut pas être modifié dans le temps) alors que l'ensemble variable *Components* est un sous-ensemble de $COMPONENTS$ et représente les instances qui sont utilisées à un instant t. Ce sont les instances qui ont été créées (et respectent donc les préconditions de cette opération) mais qui ne sont pas encore détruites.

B est basé sur un ensemble de concepts et de notations qui peuvent parfois être utilisés pour exprimer des besoins de façon similaire mais cachent en fait des nuances et impactent différemment la preuve. Dans les paragraphes suivants, nous allons décrire certaines des nuances que nous avons expérimentées dans « l'art » d'écrire une spécification en B.

Utilisation des ensembles variables versus constants Il n'est utile de déclarer des ensembles variables que pour des entités qui « varient », c'est-à-dire qui sont modifiées par les opérations de la machine B. Dans le cas contraire, cela alourdit inutilement l'invariant de machine. Ainsi, nous avons défini l'ensemble constant $IMPLANTATIONS$ pour représenter les implantations de composants mais pas l'ensemble variable *Implantations* car on peut considérer que $IMPLANTATIONS$ contient toutes les implantations à notre disposition pour créer de nouveaux templates.

Lorsqu'on modélise une association entre deux éléments, il est préférable d'utiliser des fonctions et relations sur des ensembles constants *sets* ou *constants* en ajoutant, si nécessaire, le prédicat qui restreint le domaine ou le codomaine de manière explicite plutôt que d'utiliser des fonctions et relations sur des ensembles variables *variables*. D'une part, l'information de typage apparaît plus clairement et se prouve automatiquement. D'autre part, lorsqu'on ajoute explicitement les contraintes de domaine et de codomaine, on voit mieux ce qu'il faut écrire dans les opérations pour les préserver.

Par exemple, *HasTemplate* est la fonction qui lie les composants à un template. Tous les composants créés (donc appartenant à l'ensemble variable *Components*) doivent être associés à un template. Ceci peut se traduire par l'utilisation d'une fonction totale (symbole - ->) :

$HasTemplate : Components \text{ - -> } Templates$

Quand on créé un nouveau composant *newcomponent*, il faut le rajouter à l'ensemble *Components*. Il faut alors prouver que *HasTemplate* est du type :

$HasTemplate : Components \vee \{newcomponent\} \text{ - -> } Templates$

Ceci n'est pas simple. Par contre, si on définit la fonction sur les ensembles constants $COMPONENTS$ et $TEMPLATES$, il n'y a plus ce problème car on considère tous les composants potentiels même ceux qui ne sont pas créés (appartenant à $COMPONENTS - Components$). De ce fait la fonction devient partielle (symbole +->) puisque les composants non encore créés n'ont pas besoin d'avoir une image par *HasTemplate*.

$HasTemplate : COMPONENTS \text{ +-> } TEMPLATES$

A présent, il est nécessaire de spécifier que seuls les composants créés doivent être associés à un template. Il faut alors rajouter une contrainte suivante sur le domaine de la fonction *HasTemplate* :

$dom(HasTemplate) = Components$

Dans le cas présent, l'égalité est indispensable car tous les composants créés sont issus d'un template et doivent donc être associés à un template. Dans d'autres cas, l'inclusion ensembliste suffit et sert à représenter les associations de type "0..*".

Simplification de notations La notation image $r[v]$ prend un ensemble v en paramètre et retourne un ensemble. S'il s'agit d'une relation, il n'y a pas moyen de faire autrement. Avec les fonctions, on peut écrire plus simplement $f(i)$ qui prend en paramètre un élément appartenant au domaine de la fonction f et renvoie un élément appartenant au codomaine de f. Avec cette notation, il faut s'assurer que les arguments des fonctions sont dans les domaines (le prouver), alors que ce n'est pas nécessaire quand on prend une image. Aussi, il est préférable d'utiliser tout le temps la notation image pour éviter d'avoir à « typer » toutes les variables.

Des exceptions à cette recommandation concernent les cas où on ne peut pas manipuler le résultat comme un ensemble. Par exemple, la fonction *HasGenericRoles* renvoie la séquence représentant les rôles d'un schéma d'adaptation. Dans ce cas précis, on ne peut pas utiliser la notation image car elle renvoie un ensemble contenant une séquence, ce qui empêche d'utiliser les opérations de séquence directement.

Avantage d'une traduction manuelle par rapport à une traduction automatique Nous avons vu que certains concepts et notations peuvent être utilisés pour exprimer des contraintes similaires. Cependant, du fait que ces concepts et notations ne sont pas équivalents en terme de preuve, il n'est pas très bon de vouloir traduire automatiquement une spécification UML / OCL en B à l'aide de règles toutes faites. En effet, le traducteur ne va pas faire de différence entre une classe qui peut être représentée par un ensemble constant uniquement et une autre qui doit être représentée à la fois par un ensemble constant et un ensemble variable par exemple. Pour simplifier la preuve, il est indispensable de tirer partie des spécificités du formalisme offert par B et de jouer sur ses nuances, ce qu'une transformation automatique ne permet pas. D'autre part, il n'existe pas d'isomorphisme parfait entre les concepts orienté objet utilisés par UML et les concepts sur la théorie des ensembles utilisés par B. Une spécification obtenue par transformation automatique contient donc certains détails qui vont complexifier le processus de validation inutilement.

Apport de B dans une approche MDE Il ne faut pas non plus chercher à tout traduire. Dans notre cas, B permet de tirer avantage du fait que notre modélisation se découpe en un modèle abstrait et des modèles requis. Les modèles requis ne représentent en fait qu'une abstraction des concrétisations possibles de ceux-ci. Les données constantes *constants* d'une machine représentent ce qui est acquis, qui n'a pas besoin d'être prouvé. Par exemple, la constante $CANPLAYROLE$ représente l'ensemble des couples de composants et de rôles qui satisfont cette relation. Cela nous permet de ne pas tout décrire. De ce fait, contrairement à USE, nous n'avons pas besoin de choisir un modèle de contrats particulier pour faire la preuve. De plus, cela réduit le nombre d'obligations de preuve. Le formalisme B suit totalement la philosophie MDE puisqu'il permet de ne pas prouver les modèles requis mais de les utiliser dans la preuve comme un ensemble de prédicats.

Méthodologie de preuves

La méthode B permet de garantir la conformité des opérations d'une machine vis-à-vis de son invariant en utilisant les obligations de preuve. Prouver les obligations de preuve d'une machine nous permet d'avoir deux types de retour sur la correction de la spécification UML / OCL.

- L'*obligation de preuve pour l'initialisation* permet de détecter la contradiction entre différents invariants OCL (inconsistance des propriétés).
- Les *obligations de preuve pour les opérations* permettent de déterminer les préconditions OCL d'opérations UML ne satisfaisant pas un invariant de classe (non complétude des contraintes vis-à-vis d'une propriété).

Si la preuve de deux propriétés de manière indépendante réussit et que la preuve de deux propriétés simultanément échoue alors on peut dire que les contraintes d'une propriété interfèrent avec celles de l'autre propriété (inconsistance des contraintes). Pour détecter ces trois types d'erreurs de modélisation, nous avons choisi de prouver deux propriétés. Les propriétés P_{1a} sur l'adéquation des rôles vis-à-vis de l'implantation et P_3 sur la consistance des assemblages ont été sélectionnées car toutes les deux concernent le typage des composants.

En B, la technique de preuve requiert que l'utilisateur, en plus d'être un expert en spécification, soit également un expert dans l'utilisation de cette technique de preuve. En effet, la vérification des obligations de preuve est confiée à un prouveur de théorèmes, mais dès que cet outil n'arrive plus à progresser seul vers la solution, il fait appel à l'utilisateur. En pratique, la vérification par preuve est rarement obtenue de manière automatique. Dans notre cas, comme nous avons inclus des éléments par étape et que nous avons pris soin de traduire « intelligemment » les éléments de la spécification UML, la preuve n'a pas été trop compliquée.

Validation de la machine B vis-à-vis des propriétés P_{1a} et P_3 La spécification en B des propriétés P_{1a} et P_3 a généré 71 OPs dont :

- 57 ont été prouvées automatiquement avec le prouveur automatique force 0.
- L'OP `instantiateTemplateFromFacets.6` a été prouvée automatiquement avec le prouveur automatique force 1.
- L'OP `createTemplateFromFacets.6` a été prouvée interactivement avec le prouveur de prédicats en prenant en compte uniquement les hypothèses locales.
- 8 OPs (`createTemplateFromFacets.5`, `instantiateTemplateFromFacets.7`, `instantiatePattern.3`, `instantiatePattern.6`, `instantiatePattern.7`, `instantiatePattern.8`, `instantiatePattern.10` et `removeInstance.10` ont été prouvées automatiquement avec le prouveur de prédicats en prenant en compte les hypothèses dépendantes du but.
- Les trois OPs `instantiatePattern.11`, `removeInstance.11` et `removeInstance.12` ont été prouvées interactivement avec le prouveur de prédicats en prenant en compte les hypothèses locales et respectivement l'hypothèse `H4`, `H5` et `H6`.

L'OP restante (`instantiateTemplateFromFacets.8`), liée à la préservation de la partie de l'invariant correspondant à P_3, est prouvable avec le prouveur de prédicats en prenant en compte les hypothèses dépendantes du but seulement quand on supprime la partie suivante de l'invariant correspondant à P_{1a} notée $\mathcal{P}$ dans la suite :

```
!(c,pp1).(c:Components & pp1:PROVIDEDPORTS & pp1:HasProvidedPorts[HasConcreteRoles[{c}]] =>
#pp2.(pp2:PROVIDEDPORTS & pp2:(HasProvidedPorts2[HasRoles[HasTemplate[{c}]]]
                               \/HasAdaptationPorts[HasConcreteRoles[{c}]]) &
pp1|->pp2 : EQUALS))
```

Dans le cas présent, il n'y a pas de conflit entre P_{1a} et P_3 au niveau de l'opération `instantiateTemplateFromFacets`, les ensembles sur lesquels portent respectivement les deux propriétés n'étant pas les mêmes. En fait, lorsque le prouveur de prédicats tente de prouver le but en calculant automatiquement les hypothèses dépendantes du but, tous les prédicats de la machine B qui ont des termes communs avec le but sont sélectionnés. Or certaines de ces hypothèses ne sont pas nécessaires pour établir le but. Bien qu'elles ne jouent pas un rôle « actif » dans la preuve, elles peuvent la parasiter et allonger son temps de calcul.

En mettant un petit nombre d'hypothèses dans l'opération `instantiateTemplateFromFacets` et en ajoutant une par une, tant que le prouveur n'atteint pas le but, nous avons détecté que c'était la partie de l'invariant $\mathcal{P}$ qui parasitait la preuve. Il suffit donc manuellement de supprimer cette partie de l'invariant pour que l'OP `instantiateTemplateFromFacets.8` soit prouvée. Lorsqu'on ajoute $\mathcal{P}$, le prouveur ne demande pas de prouver à nouveau cette OP. Or, si l'extension de la machine que l'on fait modifie le but à prouver, la preuve est automatiquement invalidée. Pour ça, l'atelier B est très précis et s'il n'invalide pas une preuve, c'est bien qu'elle est préservée par le nouveau système. Ceci confirme bien que $\mathcal{P}$ n'invalide pas la preuve et n'est donc pas une hypothèse à considérer dans la preuve pour l'OP `instantiateTemplateFromFacets.8`.

Des contraintes OCL vers des expressions B adéquates pour la preuve De manière générale, pour préserver un bout de l'invariant I(E) concernant l'ensemble E, le plus simple est de mettre, dans la partie précondition d'une opération, une condition de façon à ce que l'invariant soit toujours vrai sur la partie de l'ensemble modifié qui est conservée. Par exemple, si on a un bout d'invariant noté I, alors :

- Si une opération fait E := E $\vee$ E', il faut avoir comme précondition I(E'). Ce qui correspond assez bien à l'esprit de nos contraintes OCL.
- Par contre, si une opération fait E := E - E', dans ce cas, ce qui marche bien avec B, c'est de mettre comme précondition I(E-E'). Dans ce cas de figure, on s'éloigne plus de la façon dont sont exprimées les contraintes OCL.

Par exemple, considérons le bout d'invariant correspondant à la propriété de sûreté P_3. Pour l'opération `removeInstance`, la précondition suivante (notée H6 dans l'annexe D) permet de préserver l'invariant :

```
!ep.(ep:EMITTEDPORTS & ep:HasEmittedPorts[HasConcreteRoles[Components]] =>
#pp.(pp:PROVIDEDPORTS &
pp:(HasProvidedPorts -
    UNION ea.(ea:ELEMENTARYADAPTATIONS &
              ea:HasElementaryAdaptations[HasPattern[{adaptationinst}]] &
              HasType[{ea}] = {0} |
    HasConcreteRoles[{HasParticipants(adaptationinst)(ApplyPosition(ea))}]*
    HasPointcut[{ea}]))[HasConcreteRoles[HasTarget[{ep}]]] &
HasPortToEmit[{ep}]*{pp} <: FILTERS))
```

Cette condition signifie que à tout port émis *ep* contenu dans les rôles des composants, correspond un port fourni d'un rôle du composant ciblé par *ep* qui n'est pas supprimé par l'opération en cours. Dans la contrainte OCL, ceci est formulé à l'envers : aucun port fourni supprimé dans l'opération en cours n'est utilisé par les port émis contenus dans les rôles des composants. Mais l'équivalent en B de cette formulation de la contrainte n'est pas utilisable directement pour préserver P_3 vis-à-vis de l'opération `removeInstance` via le prouveur de prédicats en prenant en compte les hypothèses dépendantes du but. Bien qu'il soit possible d'établir des lemmes pour passer d'une formulation à une autre (si les deux sont bien équivalentes), cela suppose de faire un peu de preuve interactive.

Evaluation

Les obligations de preuve permettent de localiser dans les opérations de la spécification où des ensembles faisant partie de l'invariant de la machine sont modifiés. Ces obligations de preuve nous contraignent à vérifier que l'ensemble des préconditions d'une opération permet bien d'établir l'invariant après l'opération. En particulier, les six obligations de preuve que nous avons prouvées interactivement ont pu l'être car nous avons ajouté de nouvelles hypothèses dans la partie précondition des opérations `createTemplate` et `instantiateTemplate`. B nous a donc permis de corriger notre spécification en renforçant les préconditions.

Les contraintes visant à assurer les propriétés de sûreté P_{1a} et P_3 ont été prouvées. Le même travail devrait être effectué pour les contraintes relatives aux autres propriétés. Cependant, certaines propriétés (P_{1b}, P_{4c} et P_5) ne peuvent être directement exprimées dans l'invariant B. Une première solution consiste à étendre le modèle. Pour P_{1b}, il suffit de différencier l'ensembles des ports fournis initiaux d'un composant et l'ensemble courant des ports fournis d'un composant. Pour P_5, il est nécessaire de modéliser la relation d'appel entre méthodes. Cette démarche d'extension complexifie le modèle alors que l'apport de cette extension ne concerne que la preuve.

Une autre approche s'inspire des travaux autour du model-checking (voir chapitre 3) qui permettent de décrire les propriétés de vivacité (dont la détection de cycles) en utilisant des logiques temporelles telles que CTL ou LTL. Ainsi, il est possible d'exprimer des invariants correspondant à P_{4c} et P_5 comme des parcours de graphes dirigés (via les ports émis) :

- Un cycle correspond à un chemin dont on traverse un certain nombre d'arcs et qui finit par passer sur un nœud déjà rencontré.
- Un point de non-déterminisme correspond à un nœud. En partant d'un nœud N_1 se séparant en deux chemins, on traverse un certain nombre d'arcs dans chaque chemin qui finissent par se rejoindre sur un nœud N_2 représentant le point de non-déterminisme.

Dans le cas de P_{1b}, l'invariant peut s'exprimer en indiquant que pour tout $t > t_0$, E_{t0} est inclus dans E_t où E_{t0} dénote l'ensemble des ports fournis à l'instant t_0 et E_t dénote l'ensemble des ports fournis à l'instant t. La formulation de ces invariants nécessite de faire référence à la notion de temps. Une perspective dans ce domaine consisterait à évaluer s'il est possible d'utiliser les modalités *leads to* et *until* de B événementiel [2], une extension au formalisme B pour la prise en compte de propriétés dynamiques, pour exprimer P_{1b}, P_{4c} et P_5 en tant qu'invariants.

7.3 Synthèse

Pour assurer la sûreté des adaptations, nous avons défini un modèle de sûreté d'adaptations basé sur l'expression de propriétés de sûreté. UML [91] a permis d'établir rapidement et facilement une première modélisation. Tous nos besoins en terme d'expression des propriétés de sûreté ont pu être formalisés avec OCL [120]. Cette notation nous a aidé à clarifier nos besoins et à lever les ambiguïtés.

Une fois le modèle formalisé, nous l'avons validé. Pour cela nous avons eu recours à l'outil de simulation USE pour établir la correction de notre spécification. De par notre expérience, nous pouvons dire que la simulation est utile pour débuter une spécification avec OCL. Cette technique permet de détecter rapidement les erreurs grossières souvent très nombreuses en début de spécification. Elle permet de ne pas avoir recours au prototypage, plus long à mettre en place, même si finalement les deux solutions conduisent aux mêmes possibilités en terme de correction : le test.

Cependant, même si tester un programme pour s'assurer, dans un certain nombre de cas extrêmes, qu'il réagit conformément à ce qu'on attendait est une méthode importante et incontournable de validation, elle n'est pas suffisante pour réellement prouver de manière exhaustive que le programme est correct. Accumuler les présomptions n'est pas prouver. Cette approche n'a été qu'une première étape puisque l'exactitude de la spécification est garantie seulement vis-à-vis des états analysés. Aussi avons-nous complété la validation de la modélisation en utilisant la technique de preuve par théorème avec B [1] qui nous a permis de lever certains doutes sur des points sensibles de la spécification.

Cette expérience soulève plusieurs questions au niveau IDM en général. Un modèle doit être vérifié afin que les erreurs ne se retrouvent pas répercutées au niveau des plates-formes. Comment s'assurer que les modèles proposés sont corrects ? Comment sont ils prouvés ? Par mise en œuvre ? Par extension de modèles existants prouvés ? Par développement d'environnement de développement ? Doit on comme dans les cas des programmes, procéder par tests unitaires, métriques ? Pourrait on plus souvent appliquer des méthodes formelles au niveau modèle ? Une approche IDM n'est pas complète ni utilisable sans un processus de projection. Ainsi, une fois un modèle « prouvé » : comment conserver les propriétés du modèles par projection ? Certaines approches IDM utilisent le principe de raffinement pour atteindre pas à pas les solutions escomptées. Peut on expliciter la conformité d'un modèle raffiné par rapport au modèle qu'il raffine comme un ensemble de propriétés à conserver ?

Troisième partie

Extensibilité et applicabilité du modèle Satin

« L'intelligence est l'art de fabriquer des systèmes d'abstraction en présence d'une situation et de les insérer dans cette situation »

Eugène Delacroix

« L'importance d'un concept se mesure à sa valeur opératoire, au rôle qu'il joue pour diriger l'observation et l'expérience »

François Jacob

8 Applicabilité du modèle Satin

Notre approche consiste à fiabiliser le processus d'adaptation dynamique des composants indépendamment des plates-formes à l'aide du modèle de sûreté d'adaptation Satin. Cette approche entre dans le cadre de la démarche MDA [90] et de ce fait est confrontée au besoin de projeter des modèles. En effet, un modèle décrit une solution générale à un problème donné et ne peut donc pas être réutilisé tel quel dans les plates-formes concrètes.

Un mécanisme de transformation doit alors être défini conjointement au modèle. Comme nous l'avons vu au chapitre 5, l'approche traditionnelle recommandée par l'approche MDA consiste à projeter complètement le modèle en décrivant les règles de transformation liant les éléments du modèle abstrait et les éléments du modèle de la plate-forme cible. Une alternative à l'approche par projection proposée dans cette thèse consiste à mettre en œuvre un service autour de ce modèle en le rendant opérationnel (disponible à l'exécution). Le modèle n'est pas projeté dans la plate-forme et certaines des données de l'application s'exécutant sur la plate-forme cible sont « remontées » vers le modèle via le service.

La section 8.1 présente l'approche de transformation par projection du modèle Satin dans les plates-formes. La section 8.2 présente la solution adoptée dans cette thèse pour la mise en œuvre du modèle : la conception d'un service de sûreté au dessus du modèle Satin. Enfin, la section 8.3 met en évidence les apports de l'approche service.

8.1 Projection du modèle Satin

La section 8.1.1 décrit les principes de la projection. La section 8.1.2 illustre la projection pour certaines plates-formes étudiées dans le chapitre 2.

8.1.1 Principes de la projection

Le modèle Satin a été défini, dans le chapitre 5, comme une intégration de modèles requis (`ContractExtension` et `AdaptationExtension`) à un modèle fourni (le modèle abstrait `SatinBase`). Or, nous avons vu dans ce même chapitre que les modèles fournis et requis ne se projettent pas de la même manière. Par conséquent, la projection du modèle Satin se fait en deux étapes : la projection du modèle abstrait et la projection des modèles requis.

Projection du modèle abstrait

La projection du modèle abstrait est double. Elle concerne la transformation d'entités liées au processus d'adaptation et des contraintes OCL permettant de préserver les propriétés de sûreté (présentées dans le chapitre 7). La projection du modèle abstrait consiste alors à :

- identifier à quel(s) élément(s) de la plate-forme cible correspond chaque élément du modèle abstrait,
- déduire sur quel(s) élément(s) de la plate-forme se rattachent les contraintes préservant chaque propriété.

La projection du modèle abstrait n'est ni directe ni totale. Elle n'est pas directe car le but n'est pas de projeter les éléments du modèle abstrait mais les contraintes qui y sont rattachées. En effet, l'objectif n'est pas de fournir à la plate-forme de nouvelles adaptations mais de vérifier que les adaptations effectuées par la plate-forme ne remettent pas en cause le bon fonctionnement de l'application adaptée. Ainsi, les éléments du modèle abstrait sont projetés uniquement pour localiser le lieu de projection des contraintes.

Elle n'est pas totale puisque certains éléments du modèle abstrait correspondent à aucun élément dans certaines plates-formes. Il n'y a pas de projection explicite de l'élément donné dans le modèle cible. Elle n'est pas totale également dans le sens où toutes les propriétés ne sont pas nécessairement projetées. En effet comme nous l'avons mentionné dans le chapitre 5, suivant la plate-forme visée et les types d'adaptation élémentaires que la plate-forme prend en charge, certaines propriétés de sûreté n'ont pas besoin d'être vérifiées. Ainsi, seules les contraintes correspondant aux propriétés à vérifier sont projetées.

Projection des modèles requis

Nous avons vu en 5 qu'un modèle requis sert de **contrat pour la projection** puisqu'il permet localiser des points de collaboration avec la plate-forme. Suivant que le contrat est rempli ou non, la projection est différente.

- Si la plate-forme visée fournit un modèle compatible proposant des opérations équivalentes à celles du modèle requis, on projette seulement les appels aux opérations requises par Satin et fournies par la plate-forme.
- Sinon, on raffine par substitution le modèle requis par un modèle fourni conforme qui projette les éléments manquants (y compris les opérations) à la plate-forme et on projette les appels aux opérations requises générées.

Notons que suivant le sous-ensemble des propriétés qui doivent être vérifiées pour une plate-forme donnée, toutes les opérations des modèles requis n'ont pas besoin d'être projetées. Par exemple, si la propriété P_{4c} sur les points de non-déterminisme n'a pas besoin d'être vérifiée pour une certaine plate-forme p alors l'opération `ElementaryAdaptation.withoutOrdering` du modèle requis d'adaptations élémentaire n'est pas projetée. En particulier, la plate-forme p n'a pas d'obligation dans ce cas de fournir un modèle implémentant cette opération. Le contrat de projection est donc différent suivant les propriétés de sûreté considérées.

8.1.2 Exemple de projection vers les plates-formes OpenCCM, Julia et Noah

Nous avons choisi d'étudier la projection vers les plates-formes OpenCCM [72], Julia [21] et Noah [17] pour leurs différences en terme 1) d'architecture, 2) de types d'adaptation supportés, et 3) de mécanismes de mise en œuvre des adaptations.

Elément de projection du modèle de base vers les plates-formes OpenCCM, Julia et Noah

Comme nous l'avons précisé auparavant, toutes les propriétés de sûreté ne sont pas nécessairement projetées dans chaque plate-forme. Le tableau 8.1 décrit quelles propriétés doivent être projetées pour chacune des trois plates-formes étudiées. Le symbole × indique que la propriété doit être projetée. La propriété P_0 n'a pas besoin d'être projetée en Noah car la plate-forme ne permet pas le remplacement explicite de composants. La propriété P_{1b} n'est pas projetée dans ces trois plates-formes car aucune d'entre elles ne permet le retrait de fonctionnalités. La propriété P_2 n'est pas projetée en Noah car la plate-forme n'est pas contrainte par l'utilisation de facettes pour accéder aux composants. La propriété P_{4a} n'est pas projetée en OpenCCM car la plate-forme inclut la notion de point de vue, ce qui fait qu'une fonctionnalité peut apparaître dans deux facettes différentes et avoir un comportement différent suivant la facette d'appel. La propriété P_{4b} n'est pas projetée vers OpenCCM et Julia car ces plates-formes ne permettent pas la composition comportementale. La propriété P_6 n'est pas projetée dans OpenCCM et Julia car les coupes d'adaptation de ces plates-formes sont closes. Elle n'est pas projetée dans Noah car le temps d'adaptation est linéaire dans cette plate-forme : une adaptation n'a d'impact qu'au moment où elle est appliquée.

Propriété	OpenCCM	Julia	Noah
P_0 : Conservation du contexte d'utilisation	×	×	
P_{1a} : Adéquation des rôles vis-à-vis de l'implantation	×	×	×
P_{1b} : Conservation des fonctionnalités de base			
P_2 : Garantie de visibilité	×	×	
P_3 : Consistance des assemblages	×	×	×
P_{4a} : Uniformité comportementale		×	×
P_{4b} : Compatibilité des adaptations			×
P_{4c} : Points de non-déterminisme	×		×
P_5 : Cycles	×	×	×
P_6 : Rétro-activité des adaptations			

TAB. 8.1 – Propriétés à projeter dans chaque plate-forme

La projection du modèle abstrait vers OpenCCM, Julia et Noah est illustrée dans les tableaux 8.2 et 8.3. Le symbole ? indique qu'il n'y a pas de projection explicite de l'élément donné ou de la propriété dans le modèle cible, le symbole – signifie que la propriété n'a pas besoin d'être projetée.

Méta Elément	OpenCCM	Julia	Noah
Template	Home	Template	EJBHome
Role	IDL Type	InterfaceType	EJBObject
Implantation	CIF descriptors	Primitive component	Session / Entity Bean
Component	Component	Composite component	Noah object
Provided Ports	CCM facet	Server interface	EJBObject's methods
Emitted Ports	CCM receptacle	Client interface	?
Adaptation Ports	?	Interceptors, controlers	Interacting methods
Adaptation pattern	?	?	Interaction pattern
Adaptation instance	?	?	Interaction

TAB. 8.2 – Synthèse sur la projection des éléments du modèle abstrait

Bien que la projection de la plupart des éléments du modèle abstrait soit identifiée, le mécanisme de projection n'en reste pas moins difficile. En effet, l'écriture de règles de transformations des éléments du modèle abstrait est faisable mais reste intimement liée à l'architecture des plates-formes sous jacentes.

Propriété	OpenCCM	Julia	Noah
P_0	Component	Component	–
P_{1a}	Home, ? CCM facet ?	Template, ? ContentControler ?	EJBHome, interaction pattern
P_{1b}	–	–	–
P_2	? CCM receptacle ?	? BindingControler ?	–
P_3	? CCM receptacle ?	? BindingControler ?	Interaction pattern, interaction
P_{4a}	–	Template	EJBHome
P_{4b}	–	–	Interaction pattern
P_{4c}	?	–	Interaction pattern
P_5	?	?	Interaction pattern
P_6	–	–	–

TAB. 8.3 – Synthèse sur la projection des propriétés

D'une part, la transformation des éléments du modèle abstrait implique un choix lorsque les concepts n'existent pas dans la plate-forme cible. Par exemple, comment projeter une propriété dont certaines contraintes sont associées au concept de schéma d'adaptation alors que celui-ci n'est pas explicitement projeté dans Julia. Dans ce cas, il faut choisir un ou plusieurs éléments de la plate-forme cible sur lesquels projeter les contraintes. Cependant, la projection du même concept pour deux propriétés peut différer. Prenons les propriétés P_{1a} sur l'adéquation des rôles vis-à-vis de l'implantation, P_2 sur la garantie de visibilité et P_3 sur la consistance des assemblages qui sont en autre préservées par des contraintes sur les opérations de la classe `AdaptationPattern`. Les éléments les plus appropriés sur lesquels peut s'appuyer la projection de P_{1a} sont les contrôleurs de contenu (ContentControler). Par contre, les éléments les plus appropriés sur lesquels peut s'appuyer la projection de P_2 et de P_3 sont les contrôleurs de liaison (BindingControler). Dans le cas de la propriété P_5, on ne sait même pas faire ce choix.

D'autre part, la transformation des propriétés est surjective. En effet, les contraintes associées aux propriétés de sûreté étant décrites sur plusieurs éléments du modèle abstrait, elles se retrouvent éclatées par transformation à plusieurs niveaux dans les plates-formes cibles.

Elément de projection des interactions modèle abstrait-modèles requis vers les plates-formes

Les tableaux 8.4 et 8.5 décrivent la projection des interactions du modèle abstrait avec les points de collaboration des modèles requis `ContractExtension` et `AdaptationExtension`.

Opération	OpenCCM	Julia	Noah
ContractSystem.-getContract	CORBA : :Container.lookup-(in string)	java.lang.Class.forName(String)	java.lang.Class.forName(String)
ClassifierContract.-conforms	CORBA : :Object.is_a(in string), CORBA : :Object.is_equivalent(in Object)	Type.isFcSubTypeOf(Type)	Class.isAssignableFrom(Class)
FeatureContract.-conforms			

TAB. 8.4 – Synthèse sur la projection du modèle requis de contrats

Certaines interactions avec le modèle de contrats `ContractExtension` n'ont pas besoin d'être projetées vers les plates-formes étudiées. En effet, dans OpenCCM et Noah, la conformité entre types est basée sur l'héritage d'interfaces ou de classes. Dans ce cas, la conformité entre types ne dépend pas de la conformité entre types fonctionnels mais sur le graphe d'héritage. Ceci explique pourquoi l'opération `FeatureContract.conforms` n'est pas projeté en OpenCCM et Noah. En ce qui concerne Julia, la relation de conformité entre types est basé sur le polymorphisme d'inclusion et dépend donc de la conformité entre types fonctionnels. Cependant, la plate-forme Julia fournit

directement une opération permettant de déduire la conformité entre types (`Type.isFcSubTypeOf`), nous n'avons donc pas besoin de descendre au niveau des types fonctionnels pour déterminer la conformité.

Opération	OpenCCM	Julia	Noah
ElementaryAdaptation.isCompatible-With		ContentController.-checkFc()	Rule.merge(Rule)
ElementaryAdaptation.getPointcut			
ElementaryAdaptation.getType			
ElementaryAdaptation.getProperties			
ElementaryAdaptation.without-OrderingFor			
PointcutLanguage.match	Components : :Navigation.-provide_facet(in string)	BindingController.-lookupFc(String)	String.startsWith(String), String.endsWith(String), String.substring(String)

TAB. 8.5 – Synthèse sur la projection du modèle requis d'adaptations et de coupes

Certaines interactions avec le modèle d'adaptations élémentaires `AdaptationExtension` n'ont pas d'équivalent dans la plate-forme cible, ce qui peut traduire un manque de recul dans les plates-formes vis-à-vis de la modélisation des adaptations. Par exemple, il existe un équivalent pour l'opération `ElementaryAdaptation.isCompatibleWith` dans Julia (`ContentController.-checkFC` vérifie qu'il n'y ait pas d'incompatibilités de contenance, i.e. de conflits en terme de composants partagés) et dans Noah (`Rule.merge` vérifie qu'une règle d'interaction n'en masque pas une autre) mais pas dans OpenCCM. Aucun équivalent à `ElementaryAdaptation.getType` ne permet d'identifier les types d'adaptations élémentaires mis en œuvre dans chaque plate-forme bien qu'on sache clairement les identifier (cf chapitre 2), de même pour l'opération `Elementary-Adaptation.getProperties`. Ces opérations requises sont simples à mettre en œuvre. Leur absence montre peut être le besoin d'une unification des concepts dans ce domaine au travers de l'élaboration d'une API standard. C'est ce que le groupe AOP-alliance, par exemple, a commencé à faire dans le domaine de la séparation des préoccupations. Rappelons toute fois que certaines opérations n'ont pas besoin d'être implémentées lorsqu'elles ne sont pas utilisées par le sous-ensemble de propriétés à vérifier pour la plate-forme considérée.

La figure 8.1 récapitule comment les différentes dépendances entre le modèle abstrait Satin et les modèles requis de contrats et d'adaptations sont projetées vers Julia. Seules les opérations nécessaires vis-à-vis des propriétés à vérifier pour Julia sont représentées.

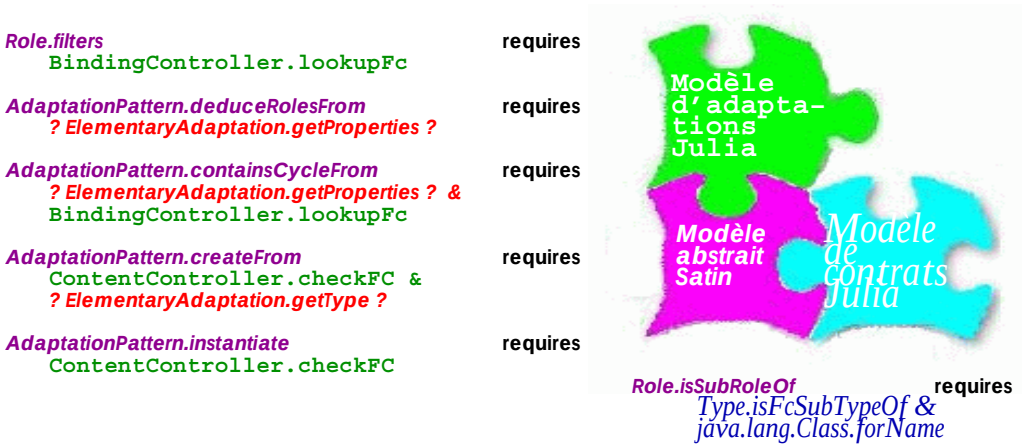

FIG. 8.1 – Intégration du modèle abstrait Satin avec les points de collaborations projetés vers Julia

A ce niveau, on peut se rendre compte que le découpage en modèle fourni et modèle requis pour expliciter la collaboration avec la plate-forme perd son sens à la projection. Du fait que les éléments du modèle abstrait sont projetés **dans** la plate-forme, on se retrouve avec des non sens. Ainsi, la notion de *rôle* qui se projette en Julia en *InterfaceType* rend la dépendance `Role.isSubRoleOf requires Type.isFcSubTypeOf` inutile.

8.1.3 Limites de l'approche par projection dans les plates-formes

Pour conclure, l'écriture de règles de transformation du modèle vers les plates-formes est coûteuse car il est nécessaire de maîtriser toutes les subtilités des plates-formes cibles. De plus, la généralisation de ces règles à différentes plates-formes parait inadéquate. Prendre le modèle comme spécification et comme méthodologie de développement afin d'intégrer directement les propriétés de sûreté à la main à chaque plate-forme semble une voie plus raisonnable. Cependant cette approche, quelle soit automatisée ou non remet en cause toutes les preuves de consistance et de complétude qui ont été faites au niveau modèle. En effet, ces preuves sont à refaire au niveau de chaque plate-forme en fonction de la mise en œuvre des contraintes et d'autres preuves peuvent également être nécessaires comme par exemple vérifier la consistance de contraintes qui ne sont pas attachées au même élément source mais qui sont, par transformation surjective, attachées au même élément cible. D'autre part, l'expression des modèles requis pour déterminer la collaboration entre le modèle Satin et les plates-formes cibles ne peut être exploité dans cette approche puisque les points de collaboration sont confondus à la projection.

La section suivante présente l'alternative à l'approche par projection du modèle dans la plate-forme cible qui a été choisie dans cette thèse pour permettre d'exploiter la collaboration des modèles.

8.2 Mise en œuvre de modèle exécutable : l'approche service

Cette approche consiste à mettre en œuvre un service autour du modèle Satin afin de le rendre opérationnel (disponible à l'exécution). Le service de sûreté permet de valider la faisabilité d'une adaptation d'une application donnée *A* en fonction des fonctionnalités offertes par les composants et des adaptations passées de *A*. Pour cela, nous avons défini un processus d'adaptation vis-à-vis duquel sont positionnées les propriétés du « Quoi », du « Quand » et du « Comment » définies dans le chapitre 5.

8.2.1 Processus d'adaptation et vérifications

Le processus d'adaptation proposé est le suivant :

Étape 1 - Création d'un schéma d'adaptation : Vérifier qu'une description d'adaptation est cohérente. Aucun sous-ensemble d'adaptations élémentaires visant à contrôler une même fonctionnalité ne doit conduire à une incompatibilité des contrôles (propriété $\mathbf{P_{4b}}$), aucun sous-ensemble d'adaptations élémentaires ne doit engendrer l'introduction d'un point de non-déterminisme ou à un cycle dans un flot d'exécution de méthodes (propriétés $\mathbf{P_{4c}}$ et $\mathbf{P_5}$) et les adaptations élémentaires doivent définir un comportement (propriété $\mathbf{P_{1a}}$). Cette étape s'appuie sur le modèle requis `AdaptationExtension`.

Étape 2 - Application d'un schéma d'adaptation

– Valider les données extraites : Extraire des données de l'application A (interfaces et implémentation du composant), une description Satin de composant et les valider à la fois par

rapport à la plate-forme cible et dans le modèle[1]. Cette étape fait ressortir un nouveau besoin en terme de modèle requis puisqu'il faut pouvoir extraire certaines données de l'application. Dans la suite, nous nommons ce modèle requis d'extraction `ExtractionExtension`.
Lors de l'extraction, il faut vérifier pour les modèles traditionnels d'objets basés sur un typage fort, par exemple, que le composant est bien défini par rapport aux implantations spécifiées. Cette validation est réalisée automatiquement par Satin si la plate-forme offre des mécanismes d'introspection (i.e. : fournit un modèle conforme à `ExtractionExtension`) ou par l'utilisateur sinon. Au niveau de la validation dans le modèle, il faut par exemple vérifier que les rôles déduits des interfaces spécifiées sont en adéquation avec les implantations spécifiées du composant (propriété $\mathbf{P_{1a}}$). Sont également réalisées les vérifications liées à la garantie de visibilité et à l'uniformité comportementale (propriétés $\mathbf{P_2}$ et $\mathbf{P_{4a}}$).

– Vérifier l'applicabilité d'un schéma d'adaptation à un ensemble de composants extraits *C* :
A cette étape, sont pris en compte les différentes adaptations déjà réalisées sur l'ensemble de composants *C*. Certaines vérifications de l'étape 1 (propriétés $\mathbf{P_{4b}}$, $\mathbf{P_{4c}}$ et $\mathbf{P_5}$) sont donc effectuées à nouveau vis-à-vis des adaptations du schéma mais aussi des adaptations antérieures réalisées sur ces composants.
Dans la mesure où l'applicabilité du schéma d'adaptation est vérifiée, les rôles des composants extraits appartenant à l'ensemble *C* sont modifiés. Cette étape effectue des vérifications supplémentaires liées à la conservation des fonctionnalités de base (propriété $\mathbf{P_{1b}}$) et à la consistance des assemblages (propriété $\mathbf{P_3}$). Cette étape s'appuie sur les modèles requis `ContractExtension` et `AdaptationExtension`.

Étape 3 - Retrait d'une adaptation : Déterminer si une adaptation peut être défaite. Nous avons vu dans le chapitre 6, que seule la propriété sur la consistance des assemblages de composants ($\mathbf{P_3}$) pouvait être mise en péril par un retrait d'adaptation. Cette étape s'appuie sur le modèle requis `AdaptationExtension`.

L'étape 1 est associée à la création d'un schéma d'adaptation correspondant aux adaptations élémentaires visées dans la plate-forme. L'étape 2 correspond à l'application d'un schéma d'adaptation à un ensemble de composants donné de la plate-forme cible et donc à construire, au niveau du modèle Satin, la représentation de ces composants. Notons que seuls les composants adaptés de l'application sont remontés au niveau du modèle Satin. L'étape 3 consiste à désappliquer un schéma d'adaptation qui a été appliqué antérieurement à un ensemble de composants. Nous ne détaillons pas l'étape de validation de l'opération de remplacement de composants car c'est une combinaison des étapes 2 et 3. Notons toute fois qu'à cette étape il faut également vérifier la propriété $\mathbf{P_0}$) pour s'assurer que les nouveaux composants sont bien substituables aux composants remplacés. Enfin, l'étape 4 associée à la destruction d'un schéma d'adaptation ne nécessite pas de vérifications supplémentaires. Chaque étape comprend 2 phases (cf. figure 8.2) :

1. une phase de requête de la part de la plate-forme (remontée d'information et analyse de cette information dans le modèle Satin) et interrogation de la plate-forme lorsque certaines vérifications font intervenir les points de collaborations définis par les modèles requis,
2. une phase de réponse du serveur du service de sûreté (redescente d'information concernant la validité de l'opération d'adaptation souhaitée vers la plate-forme).

[1] Notons que si la validité par rapport à la plate-forme cible ne peut être établie « automatiquement », l'on peut toutefois se contenter de vérifier la validité des données dans le modèle.

Il y a donc un dialogue bidirectionnel entre le serveur de sûreté et la plate-forme. Notons que les vérifications liées à chaque étape sont explicitées à titre informatif mais qu'elles ne sont pas forcément toutes réalisées puisque selon les plates-formes, certaines propriétés de sûreté n'ont pas besoin d'être vérifiées. Une configuration du service permet à chaque plate-forme de décider des propriétés de sûreté qu'elle souhaite vérifier ou non en fonction de ses besoins.

Prise en compte du « Quand »

Nous avons spécifié, dans le chapitre 5, que le moment où une adaptation devient effective (le moment à partir duquel elle a un impact sur l'exécution des composants) dépendait de la nature des modifications à effectuer. Nous complétons ici notre solution en donnant pour chaque opération d'adaptation du modèle son impact sur la détermination du « Quand ».

La prise en compte du Quand correspond à un ordonnancement du processus d'adaptation à travers l'utilisation du protocole d'application du service. Ainsi, l'étape 1 peut être effectuée à n'importe quel moment contrairement aux étapes 2, 3, et 4 qui dépendent respectivement de l'existence des étapes 1 (on ne peut pas appliquer des schémas d'adaptation non définis), 2 (on ne peut pas supprimer une adaptation non déjà créée) et 3 (on ne peut pas détruire un schémas d'adaptation s'il existe des instances d'adaptation de ce schéma). Le moment où l'instanciation d'un schéma d'adaptation peut être rendue effective dépend du type des adaptations élémentaires définies par ce schéma :

- Un port ou un rôle peut être ajouté n'importe quand.
- Un critère majeur identifiant les moments potentiellement sûrs où le comportement associé à un port p peut être effectivement modifié est qu'il n'y ait aucune communication via p entre le composant à adapter et son environnement. Le problème est que les moments satisfaisant cette contrainte sont rares. D'autre part, ceci implique au minimum d'attendre la fin du traitement de l'opération en cours d'exécution si l'opération en question est celle associée à p. Ceci n'est ni pertinent puisque cela peut prendre un certain temps, ni suffisant puisque les modifications ne doivent pas interrompre l'exécution d'une unité de travail contiguë. Une autre possibilité consiste à appliquer les modifications au prochain appel si le port à modifier est en cours d'exécution ou tout de suite sinon.
- Le moment où un composant c peut être effectivement remplacé dépend de l'item précédent.
- Le moment où l'instanciation d'un schéma d'adaptation peut être effectivement défaite dépend du moment où un port ou rôle peut être supprimé qui lui est contraint par la propriété P_3 (et plus particulièrement par la contrainte comportementale C_{15}).

Prise en compte du « Comment »

Les modifications liées à une adaptation sont répercutées au niveau du service parce qu'en terme de sûreté nous avons besoin de prendre en compte l'état des composants vis-à-vis des adaptations qu'ils ont subi. Cependant, seulement les adaptations mises en œuvre au niveau de la plate-forme ont un impact sur l'exécution des composants de l'application. Ainsi, il est primordial que les propriétés d'ACIDité que nous avons vu en 5.1 soient implémentées au niveau du mécanisme d'adaptation de la plate-forme sous forme de transactions par exemple.

Comme les adaptations sont effectuées au niveau du service, les propriétés d'ACIDité doivent être prises en compte à ce niveau également. Dans le cas contraire, la sûreté du service rendu est compromise. En effet, si pour une raison quelconque les adaptations ne sont pas toutes réalisées au niveau du service, l'état de celui-ci va être désynchronisé de l'état de l'application.

Toujours pour maintenir un état consistant entre la plate-forme et le service, toute séquence associée à une requête au service pour savoir si une adaptation est sûre ou non et la requête correspondante à sa mise en œuvre au niveau de la plate-forme doit également préserver ces propriétés.

8.2.2 Le service de sûreté comme raffinement du modèle abstrait Satin

Le service est utilisé au travers d'un protocole d'application (cf section suivante) qui permet de concrétiser le processus d'adaptation. De plus, la mise en œuvre du service fait intervenir un nouveau modèle requis pour l'extraction des données relatives à l'application. Ainsi, le service peut être défini comme un raffinement du modèle abstrait. La description Picore suivante décrit ce raffinement ainsi que le modèle requis `ExtractionExtension`.

```
metamodel ExtractionExtension
    class System
        operation System.getClassifier(String name) : Classifier
    class Classifier
        operation getFeatures() : Collection
    class Feature
        operation getName() : String
        operation getArguments() : Collection
        operation getReturn() : Classifier

metamodel SatinService refines SatinBase
                        requiring ContractExtension, AdaptationExtension, ExtractionExtension
    class Server
        // Opérations correspondant au protocole d'application
```

8.2.3 Description du protocole d'application du service de sûreté

Le protocole d'application permettant d'utiliser le service est décrit ci-après en IDL Corba. L'interface `Server` permet de manipuler le modèle Satin sans en connaître les mécanismes de manière approfondie. Les interfaces `Contract`, `Adaptation` et `Extraction` sont utilisées par le serveur pour communiquer avec la plate-forme. Ces trois interfaces correspondent à la concrétisation des opérations des modèles requis.

```
interface Server  {
  void registerComponent(in string componentName, in sequence<string> interfaceNames,
                         in sequence<string> implantationClassName)
       raises (AlreadyDefinedException, ReflectionException, SafetyPropertyViolationException,
               ServerNotFoundException);

  void unregisterComponent(in string componentName)
       raises (LifeCycleException, NotDefinedException, ServerNotFoundException);

  string createPattern(in string adaptationDescription)
       raises (AlreadyDefinedException, ParserException, SafetyPropertyViolationException,
               ServerNotFoundException);

  void removeAdaptationPattern(in string patternName)
       raises (LifeCycleException, NotDefinedException);

  string instantiatePattern(in string patternName, in sequence<string> participantNames)
       raises (AlreadyDefinedException, NotDefinedException, ReflectionException,
               SafetyPropertyViolationException, ServerNotFoundException);

  void removeAdaptationInstance(in string instanceName)
       raises (NotDefinedException, SafetyPropertyViolationException, ServerNotFoundException);

  void replaceComponent(in string oldComponent, in string newComponent)
       raises (AlreadyDefinedException, ReflectionException, SafetyPropertyViolationException,
               ServerNotFoundException);
};
```

```
interface Contract {
  boolean conforms(in ClassifierContract c1, in ClassifierContract c2);
  boolean conforms(in FeatureContract f1, in FeatureContract f2);
  ClassifierContract getClassifierContract(String name) raises (ReflectionException);
  FeatureContract getFeatureContract(String name) raises (ReflectionException);
};

interface Adaptation {
    boolean match(in String s1, in String s2);
    boolean isCompatibleWith(in ElementaryAdaptation ea1, in ElementaryAdaptation ea2);
    sequence<EmittedPort> getProperties(in ElementaryAdaptation ea) raises (ReflectionException);
    string getType(in ElementaryAdaptation ea);
};

interface Extraction {
    Classifier System.getClassifier(in string name) raises (ReflectionException);
    sequence<Feature> getFeatures(in Classifier c) raises (ReflectionException);
    string getName(in Feature f) raises (ReflectionException);
    sequence<Classifier> getArguments(in Feature f) raises (ReflectionException);
    Classifier getReturn(in Feature f) raises (ReflectionException);
};
```

Chaque opération de l'interface `Server` est associée à une étape du processus d'adaptation :

Etape 1 : Création d'un schéma d'adaptation

- `createAdaptationPattern` permet de créer un schéma d'adaptation correspondant à une description d'adaptation qui doit respecter une grammaire donnée. On ne peut pas appliquer les modifications associées à une description d'adaptation sans l'avoir enregistrée au préalable. `createAdaptationPattern` renvoie le nom du schéma d'adaptation. L'exception `AlreadyDefinedException` est levée si on tente d'enregistrer un schéma avec un nom qui est déjà utilisé. L'exception `ParserException` est déclenchée si la lecture de la description d'adaptation pose problème. L'exception `SafetyPropertyViolationException` est levée en cas de violation d'une propriété de sûreté. Comme précédemment, un problème de communication avec le serveur déclenche une exception `ServerNotFoundException`.

Etape 2 : Application d'un schéma d'adaptation

- `instantiatePattern` permet d'appliquer les modifications liées à un schéma d'adaptation à un ensemble de composants. `instantiatePattern` renvoie le nom de l'instance d'adaptation (ce nom sert par la suite à pouvoir défaire les modifications du schéma à ces composants en particulier). L'exception `NotDefinedException` est levée si on tente d'instancier un schéma d'adaptation dont le nom ne correspond à aucun schéma connu du serveur de sûreté. L'exception `SafetyPropertyViolationException` est levée en cas de violation d'une propriété de sûreté. Comme cette opération est susceptible d'appeler `registerComponent` (pour enregistrer des composants à la volée), les exceptions `AlreadyDefinedException`, `ReflectionException` et `ServerNotFoundException` peuvent être levées dans les mêmes circonstances.
- `registerComponent` permet d'enregistrer un composant au niveau du service. Cette fonctionnalité peut ne pas être utilisée explicitement. En cas d'application d'un schéma d'adaptation à un composant non encore enregistré, la fonction est appelée implicitement par `instantiatePattern`. L'exception `AlreadyDefinedException` est levée si on tente d'enregistrer un composant avec un nom qui est déjà utilisé pour un autre composant. Si un problème a lieu durant l'extraction des données une exception `ReflectionException` est déclenchée. L'exception `SafetyPropertyViolationException` est levée en cas de violation d'une propriété de sûreté. Enfin, en cas de problèmes de communication avec le serveur de sûreté, l'exception `ServerNotFoundException` est levée.

Etape 3 : Retrait d'une adaptation

– `removeAdaptationInstance` permet de défaire les modifications associées à une description d'adaptation pour un ensemble de composants. L'exception `SafetyPropertyViolationException` est levée en cas de violation d'une propriété de sûreté. L'exception `NotDefinedException` est levée si on tente de supprimer une instance d'adaptation dont le nom ne correspond à aucune instance connue du serveur de sûreté. Comme précédemment, un problème de communication avec le serveur déclenche une exception `ServerNotFoundException`.

Etape 4 : Destruction d'un schéma d'adaptation

– `removeAdaptationPattern` permet de supprimer une description d'adaptation enregistrée auprès du service sous un nom donné. L'exception `NotDefinedException` est levée si on tente de supprimer un schéma d'adaptation dont le nom ne correspond à aucun schéma connu du serveur de sûreté ou `LifeCycleException` si le schéma a encore des instances en fonctionnement.

Etape hybride (2 + 3) : Remplacement de composants

– `replaceComponent` permet de remplacer un composant par un autre dans l'assemblage dont le composant à remplacer fait parti. Cette fonctionnalité est implémentée comme une combinaison de `instantiatePattern` et `removeAdaptationInstance` et lève donc l'union des exceptions levées par ces deux opérations. Notons que cette opération n'a aucun effet si le composant à remplacé n'a jamais été adapté.

Etape 5 : Désenregistrement de composants

– `unRegisterComponent` permet de désenregistrer un composant enregistré auprès du service sous un nom donné. L'exception `NotDefinedException` est levée si on tente de supprimer un composant dont le nom ne correspond à aucun composant connu du serveur de sûreté ou `LifeCycleException` si le composant est utilisé dans au moins une adaptation. Comme précédemment, un problème de communication avec le serveur déclenche une exception `ServerNotFoundException`.

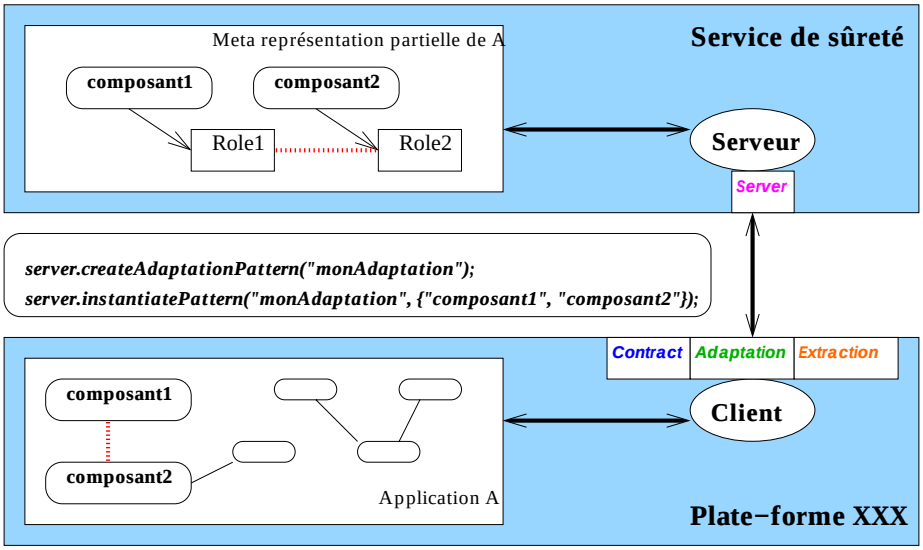

FIG. 8.2 – Architecture du service de sûreté : adaptation de composants et requêtes au service

8.2.4 Concrétisation du service

Dans le cas de la projection, décrite en 8.1, le modèle abstrait disparaît à l'exécution. Dans l'approche service, le modèle abstrait est concrétisé et existe à l'exécution au niveau du service. Le protocole d'application (les « interactions » entre le service et ses clients) du service est également concrétisé en implémentant l'interface `server`. Des modèles conformes aux modèles requis (implémentant les interfaces `Contract`, `Adaptation` et `Extraction`) sont choisis par configuration du service de sûreté. Un prototype du service a été réalisé avec les choix de concrétisation suivants :

- **Le modèle abstrait : cœur de la concrétisation du service**.
  Le prototype a été réalisé en concrétisant le modèle abstrait par des classes Java.

- **Intégration des contraintes OCL**.
  Le processus d'adaptation décrit dans la section précédente est controlé par les contraintes comportementales OCL définies dans le chapitre 7. Ces contraintes sont injectées dans le code des classes du service en utilisant la boite à outil Dresden-OCL [123]. L'intérêt d'utiliser un interpréteur OCL est que de cette manière nous n'avons pas à nous préoccuper de prouver la consistance et la complétude des contraintes vis-à-vis des propriétés étant donné que celles-ci ne sont pas concrétisées[2].

- **Mise en œuvre du Quand et du Comment**.
  Le mécanisme d'exception Java est utilisé pour que la manière d'utiliser le protocole d'application soit respectée. Les propriétés d'acidité du Comment sont mises en œuvre en utilisant des transactions Java. D'autre part, la liste des propriétés à vérifier ne peut pas être changée (c'est à dire ajouter ou supprimer des contraintes) durant le cycle de vie d'une adaptation (création, instanciation, suppression). Cette liste est modifiable uniquement quand il n'existe aucun schéma d'adaptation dans la base « active » du serveur. A chaque fois qu'on veut utiliser de nouveaux schémas, il faut faire la demande au serveur pour les créer dans la base active avec acceptation par rapport aux propriétés en vigueur à ce moment là.

- **Concrétisation du protocole d'application**.
  L'interface `server` devient une interface Java-RMI. Le prototype est actuellement utilisable à partir des plates-formes NOAH et Fractal (la section 8.2.5 montre comment utiliser cette interface dans une application / plate-forme). Chaque plate-forme doit fournir, pour chacune des interfaces `Contract`, `Adaptation` et `Extraction`, une classe d'implémentation afin d'effectuer la liaison entre le service et la plate-forme. Nous avons vu en 8.1 les correspondances entre les opérations des modèles requis et les opérations des plates-formes. Ces correspondances permettent de configurer le service vis-à-vis des modèles requis.

Rappelons que certains points de collaboration du modèle d'adaptations élémentaires n'ont pas d'équivalent dans toutes les plates-formes cibles. C'est pourquoi nous fournissons une implémentation Java du modèle `ASLAdaptation`, présenté en section 6, qui permet de décrire les adaptations avec le langage ASL et qui est compatible avec le modèle requis `AdaptationExtension`. Cette implémentation sert de configuration par défaut du modèle requis d'adaptations.

Enfin, pour savoir comment configurer l'extraction des données, il faut étudier la projection du modèle requis d'extraction des données `ExtractionExtension`. Dans le cas des plates-formes Julia et Noah, l'API de réflexion Java a été utilisée pour répondre aux besoins de ce modèle requis. Le tableau 8.6 détaille la projection des différentes opérations vers l'API Java Reflect.

[2] La validité des contraintes est conservée de par la « confiance » accordée à l'interprète.

Opérations du modèle requis d'extraction	Julia & Noah
System.getClassifier(String name)	java.lang.Class.forName(String)
Classifier.getMethods()	java.lang.Class.getDeclaredMethods()
Feature.getName()	java.lang.reflect.Method.getName()
Feature.getArguments()	java.lang.reflect.Method.getParameterTypes()
Feature.getReturn()	java.lang.reflect.Method.getReturnType()

TAB. 8.6 – Projection du modèle requis d'extraction des données vers Julia et Noah

Pour d'autres plates-formes telles que OpenCCM, l'utilisation de l'API Java Reflect n'est pas toujours possible (suivant le langage d'implantation des composants). Dans le cas d'OpenCCM, il faut passer par l'API du référentiel des interfaces (IFR) qui fournit des opérations d'introspection. A partir de l'opération `get_interface` de l'interface `CORBA : :Object`, il est possible d'obtenir la référence de l'objet du référentiel décrivant l'interface du composant afin de découvrir quelles sont les opérations et le type de leurs paramètres.

La section suivante explique comment utiliser le service de sûreté.

8.2.5 Utilisation du service de sûreté

Pour utiliser le service, il faut expliciter où et comment appeler le service de sûreté dans la plate-forme. Deux approches sont possibles suivant l'acteur qui effectue l'intégration du service : soit introduire les appels au niveau d'une application particulière par le développeur d'applications soit automatiser l'introduction des appels directement dans la plate-forme au niveau de l'API d'adaptation par le développeur de plate-forme.

Introduction automatique des appels au service par le développeur de plate-forme

Pour automatiser l'utilisation du service de sûreté, les appels au service de sûreté doivent être placés directement dans l'API d'adaptation ou les mécanismes d'adaptation de la plate-forme. Dans Noah, les appels au service de sûreté sont placés dans les primitives d'adaptation, c'est à dire dans `registerInteractionPattern`, `instantiateInteraction`, `removeInteraction`. Dans Jac [97], ils pourraient se positionner dans le tisseur. Dans Julia, la définition de trois contrôleurs de sûreté spécialisant respectivement `BindingController`, `ContentController` et `LifeCycleController` permettrait de gérer les appels au service.

Dans ce dernier cas, et de manière générale, pour les plates-formes mettant en œuvre les adaptations de manière programmative, il est difficile d'identifier les adaptations élémentaires interdépendantes. En effet, chaque adaptation élémentaire correspond une opération de l'API d'adaptation fournie par la plate-forme. Or, nous avons vu au chapitre 6 que vérifier l'applicabilité d'un schéma n'est pas équivalent à vérifier l'applicabilité des adaptations élémentaires du schéma indépendamment. Dans Julia, chaque contrôleurs « écoutent » les types d'adaptations élémentaires qu'ils prennent en charge et sont notifiés à chaque fois qu'une adaptation élémentaire est entamée. Aucune information ne permet de déterminer si les notifications successives concernent une même unité d'adaptation. D'autre part, il peut être nécessaire de coordonner les contrôleurs si des adaptations de type différent sont interdépendantes.

Cette solution rend l'acquisition du service transparente vis-à-vis de l'application mais difficile à mettre en place vis-à-vis de la prise en charge des adaptations. Le service de sûreté est configuré par défaut avec comme modèle d'adaptations élémentaires le modèle `ASLAdaptation` puisque la plus part des plates-formes ne fournissent pas les opérations requises par le modèle d'adaptation. La traduction des adaptations de la plate-forme en descriptions ASL doit nécessairement être effectuée de manière automatique. En effet, il ne serait pas très judicieux de demander une intervention humaine à chaque fois que le service est sollicité surtout s'il s'agit d'une plate-

forme auto-adaptative. Ainsi, un nouveau modèle requis est défini pour gérer la transformation des adaptations de la plate-forme en descriptions ASL. Ce modèle requis a pour mission de mettre en correspondance la grammaire ASL et la grammaire (si adaptation descriptive au niveau de la plate-forme) ou l'API d'adaptation de la plate-forme. On suppose donc avoir à disposition une fonctionnalité `translateToASL` qui à partir d'une description d'adaptation produit une description sous forme d'un schéma d'adaptation. Cette fonctionnalité est déléguée à la plate-forme.

```
metamodel TranslationExtension
    class Translate
        operation translateToASL(desc) : AdaptationPattern
```

Notons qu'étant donné que le service peut être configuré pour choisir les propriétés à vérifier, il est également possible de le rendre complètement « muet ». En effet, une intégration directe dans la plate-forme ne doit pas obliger tous ses utilisateurs de passer par le service de sûreté. Au niveau des performances, le coût d'un appel au service lorsque celui-ci n'a pas de propriétés à vérifier est négligeable.

La section suivante vise à intégrer les appels au service de sûreté non plus au niveau du code de définition de l'API d'adaptation mais à l'endroit où l'API est utilisée dans l'application.

Introduction des appels au service par le développeur d'application

La solution présentée dans la section précédente est parfois difficile à mettre en place surtout lorsque les adaptations sont programmées. Dans ce cas, l'identification des adaptations élémentaires interdépendantes est délicate. Lorsque l'intégration du service de sûreté ne peut pas être prise en charge par la plate-forme, l'intégration doit se faire au niveau de l'application à adapter.

Si l'on reprend les étapes du protocole d'application, le développeur d'application doit déterminer les moments d'adaptation c'est-à-dire les moments où sont effectuées les différentes requêtes d'adaptation de l'application. Aussi, les appels au service de sûreté doivent être placés dans le code de l'application juste avant les endroits où les adaptations sont effectuées. La traduction des adaptations de la plate-forme en descriptions ASL peut être gérée de manière ad-hoc en même temps que l'introduction des appels au service dans le code de l'application. L'utilisation du modèle requis `TranslationExtension` n'est pas obligatoire dans ce cas.

Pour les plates-formes mettant en œuvre les adaptations de manière programmative, comme le développeur d'application maîtrise l'intégralité du code d'adaptation, il est plus simple d'identifier l'unité d'une adaptation que dans la solution précédente. Dans ce cas, il faut déterminer quels ensembles d'appels à l'API d'adaptation font partis d'une même adaptation et ceux qui peuvent être vérifiés indépendamment et donc décrits par des schémas d'adaptations différents. Identifier les blocs d'adaptations élémentaires interdépendantes est également important pour respecter les propriétés d'acidité liées à la mise en œuvre du Comment.

Pour la plate-forme Julia [21] par exemple, il est envisageable de considérer que l'ensemble des appels à l'API d'adaptation effectués entre l'arrêt et le redémarrage d'un ensemble de composants correspond à un unique schéma d'adaptation. Cependant, cette approche ne parait pas toujours très satisfaisante car un grand ensemble de vérifications doit être effectué. Or, il suffit d'une seule violation de propriétés pour que l'ensemble des modifications soit rejeté alors qu'un sous-ensemble de ces modifications pourrait être valide et appliqué séparément du reste.

Une solution plus fine, consiste à placer l'appel à la méthode `createAdaptationPattern` du service avant un ensemble d'appels à la primitive d'adaptation élémentaire `bindCC` visant à construire un composant composite et à placer l'appel à la méthode `instantiatePattern` du service avant l'appel à la primitive d'adaptation élémentaire `instantiateFC` visant à instancier un composant (composite ou primitif).

Reprenons l'exemple du service de persistance décrit dans le manuel de référence de Fractal[20] et supposons que :

- L'interface `PersistenceItf` comporte les méthodes `save(Object)` et `load(String): Object`
- L'interface `StorageItf` comporte les méthodes `store(Object)` et `retrieve(String): Object`
- L'interface `LogItf` comporte la méthode `write(String, Object)`
- Les interfaces `CacheItf` et `ReplaceItf` comportent les méthodes `put(Object)` et `get(String): Object`
- `Object` est la classe racine du langage Java et comporte entre autre la méthode `equals(Object)`

L'ensemble des `bindCC` permettant d'assembler l'ensemble des composants pour réaliser le service de persistance se traduit par le schéma d'adaptation `persistanceService` décrit ci-après.

```
adaptationPattern persistanceService (Persistance p, Storage s, Log l,
                                      Cache c, Replace r) {
   modifyPort p.save(Object o) -> s.store(o); c.put(o); l.write(_call, o) ,
   modifyPort p.load(String cle) -> Object o = c.get(cle);
                                    if (o.equals(null)) then
                                       o = s.retrieve(cle); l.write(_call, o)
                                    else
                                       l.write(_call, o)
                                    endif ,
   modifyPort c.put(Object o) -> r.put(o) ,
   modifyPort c.get(String cle) -> r.get(cle)
}
```

Ainsi, pour valider la création d'un composite, il suffit de créer d'abord le schéma d'adaptation `persistanceService`. Si la tentative de création de schéma échoue, la création du composite peut être évitée. De même, pour valider la création d'une instance de composite, il suffit d'appliquer d'abord le schéma d'adaptation `persistanceService` à l'ensemble des composants concernés. Si la tentative d'application de schéma échoue, l'instanciation du composite peut être évitée.

En ce qui concerne les plates-formes basées sur une description des adaptations, l'identification des appels au service est plus facile puisque les adaptations élémentaires interdépendantes sont explicitées dans une même unité d'adaptation. Les appels au service précédent alors le moment où une unité d'adaptation est manipulée via l'API d'adaptation. Notons que dans ce cas, les opérations de cette API ne correspondent pas à des adaptations élémentaires.

Par exemple, pour la plate-forme Noah [17], les appels au service se situent au moment où les interactions sont définies, posées et supprimées c'est à dire à l'appel de chaque opération de l'API d'adaptation de Noah[3]. Par exemple, chaque fois qu'un schéma d'interaction est défini dans Noah, on doit d'abord contrôler via le service de sûreté que la description d'adaptation correspondant à ce schéma d'interaction est valide avant de permettre sa création.

Cette solution est simple à mettre en œuvre vis-à-vis de la reconnaissance des unités d'adaptation. Cependant, elle nécessite d'être répétée pour chaque application souhaitant bénéficier du service et n'est pas transparente. De plus, cette approche n'est envisageable que dans le contexte où le processus d'adaptation est déclenché par un utilisateur (humain) et non pas un processus automatique (cas des plates-formes auto-adaptatives).

[3] `registerInteractionPattern` permet de définir des interactions, `instantiateInteraction` permet de « lier » des entités et `removeInteraction` permet de supprimer un lien précédemment créé.

8.2.6 Exemple d'utilisation du service de sûreté par un développeur d'application FAC/Julius

Pour mieux comprendre les échanges entre le service et la plate-forme, reprenons à l'exemple de l'application d'agendas et supposons que celle-ci s'exécute sur la plate-forme FAC / Julius que nous avons étudiée dans le chapitre 2. Rappelons que l'agenda de l'équipe propose des fonctionnalités de base permettant de manipuler les rendez-vous (Figure 8.3, interface `AgendaDeBase`). L'agenda de Anne-Marie permet en plus de gérer les collisions de rendez-vous (Figure 8.3, interface `AgendaEvolue`).

```
Interface AgendaDeBase {
   void addRdv(Rdv);
   void removeRdv(Rdv);
   Rdv getRdv(Date);
}
```

```
Interface AgendaEvolue {
void addRdv(Rdv);
   void removeRdv(Rdv);
   Rdv getRdv(Date);
   boolean isFree(Rdv);
   void printError(String);
}
```

FIG. 8.3 – Interfaces respectivement fournies par *agendaAM* et *agendaEquipe*

Mise en œuvre de l'adaptation dans FAC/Julius

Supposons que le développeur d'application Anne-Marie veuille synchroniser son agenda avec celui de l'équipe. FAC offre la possibilité de modifier le comportement d'un composant en connectant simplement un composant d'aspect à un composant Fractal classique. Rappelons que ce nouveau type de composants et de connexion inter-composants permet d'allier les avantages des approches composants et aspects. Dans l'exemple, cette approche permet à Anne Marie d'arrêter la synchronisation simplement en déconnectant le composant d'aspect de synchronisation de *agendaEquipe* sans changer ce dernier.

L'adaptation consiste alors à intercepter les appels à `addRdv` sur le composant *agendaEquipe* en le connectant (via la méthode `wave`) à un composant mettant en œuvre un aspect de synchronisation et à connecter ce même composant d'aspect à *agendaAM* (via la méthode `bindFc`) pour communiquer à ce dernier les rendez-vous de l'équipe. La figure 8.4 décrit l'assemblage de ces composants.

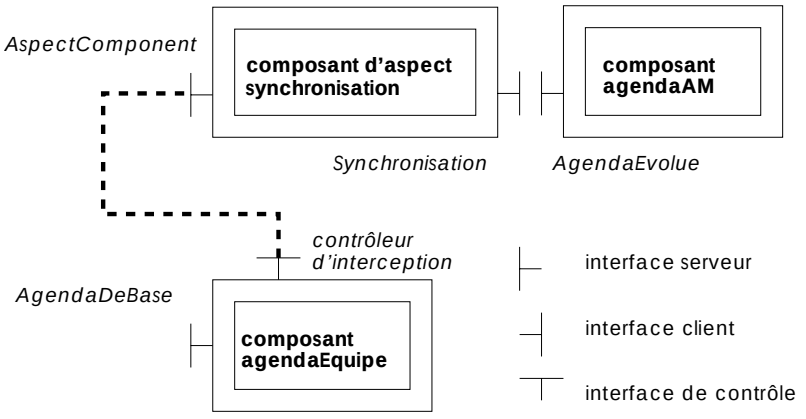

FIG. 8.4 – Architecture d'un assemblage avec un composant d'aspect de synchronisation

Pour mettre en œuvre l'aspect de synchronisation, on va créer le type de composant primitif d'aspect `synchronisationAC` permettant de modifier le comportement des composants auxquels les composants de ce type seront connectés. Le type `synchronisationAC`[4] et le code de la méthode `invoke` correspondant à l'advice de l'aspect de synchronisation sont définis ci-après par les figures 8.5 et 8.6 respectivement.

```
ComponentType synchronisationAC = typeFactory.createFcType(new InterfaceType[]{
typeFactory.createFcItfType("server1", "AspectComponent", TypeFactory.SERVER,
                            TypeFactory.MANDATORY, TypeFactory.SINGLE)});
typeFactory.createFcItfType("server2", "Synchronisation", TypeFactory.SERVER,
                            TypeFactory.MANDATORY, TypeFactory.SINGLE)});
typeFactory.createFcItfType("sync", "Synchronisation", TypeFactory.CLIENT,
                            TypeFactory.MANDATORY, TypeFactory.SINGLE)});
```

FIG. 8.5 – Définition du type de composant d'aspect de synchronisation

```
import org.aopalliance.intercept.MethodInvocation;

public Object invoke(MethodInvocation mi) throws Throwable {
  Argument arg = (Argument) mi.getMethod().getArguments().get(0);
  Object ret = null;

  if (isFree(arg)) {
    ret = proceed(mi);
    addRdv(arg);
  } else
    printError("agendas non synchronisés");

  return ret;
}
```

FIG. 8.6 – Code pour l'advice de l'aspect de synchronisation

Vérification de l'adaptation à effectuer par le service de sûreté

Pour vérifier que cette adaptation est valide dans la plate-forme il faut :

1. Enregistrer les composants à adapter au niveau du service ;
2. Créer un schéma d'adaptations à partir des adaptations élémentaires décrites dans la plate-forme ;
3. Déterminer si le schéma d'adaptation correspondant à l'adaptation visée dans la plate-forme est applicable à la représentation Satin des composants.

L'enregistrement des composants se fait en utilisant l'interface `Server`. A cette étape, le serveur de sûreté fait appel à la plate-forme via l'interface `Extraction` afin d'extraire les informations nécessaires à la représentation des composants au niveau du modèle Satin. La figure 8.7 décrit les rôles déduits des deux composants *agendaAM* et *agendaEquipe* et du composant d'aspect de synchronisation.

[4] L'interface `Synchronisation` comporte les méthodes `addRdv`, `isFree` et `printError`.

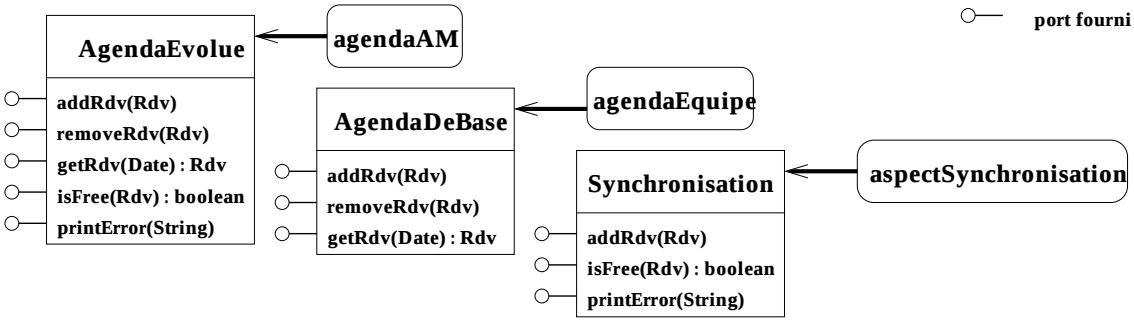

FIG. 8.7 – *agendaAM*, *agendaEquipe* et composant d'aspect de synchronisation

Une fois les informations extraites, des composants Satin correspondant à *agendaAM*, *agendaEquipe* et au composant d'aspect de synchronisation sont créés. A cette étape, le serveur de sûreté interroge la plate-forme via l'interface `Contract` pour déterminer si les rôles des composants sont en adéquation avec leur implantation vis-à-vis des règles de typage de la plate-forme.

L'adaptation au niveau du modèle Satin est représentée par le schéma d'adaptation `Synchronisation`. Dans le cas où la plate-forme ne fournirait pas une classe d'implémentation de l'interface `Adaptation` représentant le modèle requis d'adaptations, les adaptations élémentaires de la plate-forme peuvent se réécrire à l'aide du langage ASL (cf chapitre 6) comme le montre la figure 8.8. Le schéma est enregistré auprès du service via l'interface `Server`.

```
adaptationPattern Synchronisation(AgendaSynchronisant a1, AgendaSynchronisé a2,
                                  SynchronisationAspect na) {
 modifyPort a1.addRdv(Meeting m) -> if (na.isFree(m)) then
                                       a1._call(m);
                                       na.addRdv(m)
                                    else
                                       a1._call(m);
                                       na.printError("agendas non synchronisés")
                                    endif ,
 modifyPort na.isFree(Meeting m) -> a2.isFree(m) ,
 modifyPort na.addRdv(Meeting m) -> a2.addRdv(m) ,
 modifyPort na.printError(String s) -> a2.printError(s)
}
```

FIG. 8.8 – Schéma d'adaptation `Synchronisation`

Le schéma comporte quatre adaptations élémentaires de type **modification du comportement d'une fonctionnalité existante**. Le rôle déduit `AgendaSynchronisant` fournit le port `addRdv`. `AgendaSynchronisé` et `SynchronisationAspect` sont deux autres rôles déduits fournissent les ports `addRdv`, `isFree` et `printError`. Ainsi, tout composant comportant au moins un port conforme (au sens du filtrage) à `addRdv` (resp. `addRdv`, `isFree` et `printError`) peut jouer le rôle `AgendaSynchronisant` (resp. `AgendaSynchronisé` et `SynchronisationAspect`).

Evaluer si cette adaptation est possible dans la plate-forme revient à vérifier l'applicabilité du schéma d'adaptation `Synchronisation` aux trois composants Satin. Dans notre cas, nous voulons que *agendaAM* joue le rôle de l'agenda synchronisé, *agendaEquipe* celui de l'agenda synchronisant et le composant d'aspect celui de l'aspect synchronisant.

L'applicabilité du schéma est déterminée via l'interface `Server`. A cette étape, le service utilise l'interface `Adaptation` pour récupérer des informations relatives aux adaptations élémentaires et l'interface `Contract` pour évaluer si les propriétés de sûreté sont préservées. Si l'adaptation est possible (i.e. : le schéma `Synchronisation` est applicable), un assemblage est créé, le rôle de *agendaAM* est modifié pour mémoriser l'adaptation et celle-ci est réellement mise en œuvre dans la plate-forme. Si un problème intervient durant l'adaptation, les propriétés d'ACIDité qui lie le service et la plate-forme doit permettre de revenir en arrière pour assurer que l'état de l'application du point de vue des adaptations soit équivalent au niveau du service et dans la plate-forme.

8.3 Apports de l'approche service

Bien que la modélisation Satin entre dans le cadre de la démarche IDM, il n'est cependant pas envisageable de proposer des règles de transformation simple vers les plates-formes cibles vu leur diversité et leur richesse d'une part, et la complexité des contraintes OCL d'autre part. Il nous a semblé que la solution qui consiste à proposer un service de sûreté de haut niveau soit un compromis idéal pour faciliter l'intégration de la sûreté aux diverses plates-formes. De plus, la prise en charge des contraintes de sûreté se retrouve déléguée au niveau du service et l'intégration via l'appel du service permet d'identifier plus clairement en fonction des besoins de la plate-forme cible les lieux et moment des appels. La mise en œuvre d'un service permet donc de définir clairement le processus d'adaptation et ainsi de concrétiser les propriétés du Quand et du Comment.

Le service de sûreté permet de réduire à une seule fois l'étape de revalidation des contraintes contre N fois pour l'approche par projection traditionnelle lorsqu'on s'intéresse à N cibles. En effet, il suffit de prouver que la consistance et la complétude sont conservées par la concrétisation des contraintes choisie pour le service. L'avantage de cette solution est que l'apparition d'une nouvelle plate-forme ne remet pas en cause la validité de la concrétisation du service puisque celle-ci ne change pas.

Cette approche nous permet d'affirmer que l'IDM est un atout pour renforcer les travaux autour des vérifications formelles des programmes. En effet la vérification formelle de programme est une tâche lourde et de longue haleine qui doit être réitérée pour chaque évolution du programme et différente pour chaque nouveau programme. Si nous avions mis en place des vérifications de sûreté directement au niveau des plates-formes, il aurait été difficile de prouver que l'on répondait bien aux attentes et vérifiait bien les propriétés visées sans re-examiner chaque implémentation spécifique. En attachant les vérifications formelles au niveau modèle, on peut prouver le système une fois et une seule.

L'approche service semble donc être une voie prometteuse pour conserver les propriétés du modèle à l'exécution. Dans cette approche, la prise en charge des propriétés du « Quoi » se retrouve déléguée au niveau du service. Cependant elle dégrade l'efficacité dans certains cas. En particulier, pour les plates-formes à composants intégrant des moyens de vérifications tels que les contrats, le service de sûreté peut être allégé car la plate-forme peut prendre en charge les contraintes et donc proposer une implémentation plus efficace de cette validation. Par exemple, une autre concrétisation du service de sûreté utilisant le système de contrat comportemental Confract [31], permettrait de minimiser les allers-retours entre le service de sûreté et la plate-forme Julia en déportant la vérification des contraintes de sûreté au niveau de la plate-forme. Le revers de la médaille est que, dans un tel cas, on perd la validation des contraintes qui a été faite au niveau de modèle Satin : les preuves formelles doivent être refaites.

Enfin, l'applicabilité du modèle suivant l'approche service a été évalué par un prototype. Le prototype du service de sûreté est actuellement accessible aux plates-formes Noah [17] et Fractal/Julia [86]. Dans les deux cas, l'intégration est faite au niveau d'une application particulière. L'intégration dans Noah est actuellement en cours de réalisation.

« C'est au moment où un concept change de sens qu'il a le plus de sens. »

Gaston Bachelard

9

Extensibilité du modèle : prise en compte de la mobilité

DANS cette partie nous abordons la problématique de l'adéquation du modèle Satin via sa capacité en terme d'extensibilité. En effet, nous pouvons nous poser des question telles que : 1) le modèle est-il suffisamment abstrait pour prendre en compte de nouvelles formes d'adaptation ?, 2) Peut-il être facilement étendu sans pour autant le rendre trop complexe en terme de concepts ?, 3) l'ensemble des propriétés de sûreté sont-elles vérifiables pour cette extension ? Pour apporter des réponses à ces questions, nous prendrons comme domaine d'application les applications nomades et plus particulièrement la sûreté de fonctionnement des applications en présence de déconnexions. Nous proposons alors de considérer les déconnexions comme des adaptations d'assemblages de composants afin de voir si nos travaux autour de la sûreté des composants peuvent être réutilisés. Cette étude doit permettre de valider l'extensibilité du modèle, propriété indispensable pour suivre les évolutions permanentes des plates-formes et leur nombre grandissant

Notons que de nombreux travaux autour de la tolérance aux fautes traitent du problème de la déconnexion [58, 108]. Le principe de base est d'utiliser des réplicats pour assurer la continuité de service lors de déconnexions, de journaliser les opérations effectuées pendant les phases de déconnexion et enfin de réconcilier les données lors de la reconnexion. Ces travaux ont traité le problème des déconnexions principalement d'un point de vue de la consistance de l'état des données [28] et de la détection de défaillances [116]. Nous complétons ici ces travaux en nous focalisant sur la consistance des assemblages et la cohérence des communications. D'autre part, notre approche pour la sûreté de fonctionnement en environnement dynamique se basant sur la prévention (visant à empêcher l'introduction de fautes), nous intervenons donc en amont des travaux autour de la tolérance aux fautes [114].

La section 9.1 explicite les différents besoins en terme de déconnexions des applications mobiles. La section 9.2 décrit les extensions et modifications à apporter à Satin pour la prise en compte de la mobilité. Enfin, la section 9.4 conclut sur le degré d'extensibilité du modèle.

9.1 Domaine d'étude : composants et déconnexions

Les nouveaux équipements d'aujourd'hui (PDA, WiFi, ...) enrichissent les possibilités d'interconnexion, et l'avenir semble tendre vers des systèmes informatiques ubiquitaires et mobiles. L'utilisateur nomade peut alors continuer à utiliser des services applicatifs sans tenir compte de sa position. Dans un tel contexte, les applications doivent faire face au problème de la disponibilité de ces services en présence de déconnexions. Ces déconnexions peuvent être initiées par l'utilisateur qui souhaite économiser les ressources de son terminal mobile ou minimiser le coût des communications distantes, par exemple, ou bien elles peuvent être provoquées par des perturbations du réseau ou le déplacement de l'utilisateur hors de portée d'un réseau.

A partir de l'exemple développé dans la section 9.1.1, la section 9.1.2 explicite les besoins en terme d'adaptation des applications nomades. La section 9.1.3 montre comment certaines plates-formes à composants traitent la nomadicité et quels sont les apports de la modélisation et mise en œuvre d'une application mobile sous forme d'assemblages de composants.

9.1.1 Un service d'urgence construit par assemblage de composants

Dans la suite de cette partie, nous utilisons, comme exemple fil rouge, une application modélisant un service d'urgence médicale lors d'une catastrophe (accident d'avion ou de train, tremblement de terre, ...) introduite par [63]. Cette application comporte plusieurs acteurs répartis sur le terrain et dans des structures fixes. Les sauveteurs (secouristes, pompiers, ambulanciers, ...) sont éparpillés sur le lieu de l'accident et viennent au secours des patients. Les ambulances sont équipées de matériels médicaux sophistiqués pour des interventions complexes. Un centre de commandement détaché est déployé proche du lieu de l'accident afin de gérer les ambulances et les sauveteurs et communique avec les hôpitaux régionaux.

Dans la suite, nous détaillons un cas d'utilisation de l'application du service d'urgence afin de comprendre les différents enjeux d'une déconnexion.

Les patients sont couverts avec des couvertures appareillées avec des calculateurs embarqués enregistrant périodiquement des informations médicales (température, rythme cardiaque, rythme respiratoire, ...). Ces informations sont transmises via un réseau sans fil aux sauveteurs équipés d'assistants personnels numériques. Les sauveteurs peuvent également entrer manuellement les informations médicales via un éditeur spécifique.

Pour clôturer un dossier médical (via l'opération `completeReport`), les sauveteurs consultent et modifient des données, rassemblées sous la forme d'un dossier médical : donc des données sont à charger et à archiver depuis et vers le centre de commandement. Les sauveteurs peuvent également demander des expertises médicales pour les aider à établir un diagnostic (`diagnose`). Pour cela, ils communiquent par une application collaborative (le chat par exemple) avec les médecins du centre de commandement. Ils peuvent également accéder à une application de type système expert disponible sur leur PDA.

Prenons le scénario de base suivant. Le secouriste *Paul* s'occupe d'un patient dont la couverture attribuée transfère l'état du patient (`sendData`) sur le PDA de *Paul* toutes les dix minutes. *Paul* établit des dossiers médicaux qui sont ensuite transférés au centre de commandement pour archivage (`store`) dans une base de données et toutes les opérations sur les dossiers médicaux sont enregistrées (`addEvent`) dans un historique d'opérations. Enfin, *Paul* est relié via l'application de chat au centre de commandement. Une telle application peut être décrite à partir de l'assemblage de composants illustré sur la figure 9.1. Dans cet assemblage, les composants en présence sont *Paul* (une instance de composant `EmergencyAssistant`), *plaid53* (instance de `Plaid`), *irc4Paul* (instance de `Chat`), *dbms* (instance de `DataBase`) et *december* (instance de `Log`). Ils interagissent entre eux pour mettre en œuvre le scénario. Ils ne sont pas tous situés sur le même support matériel et communiquent via des réseaux différents.

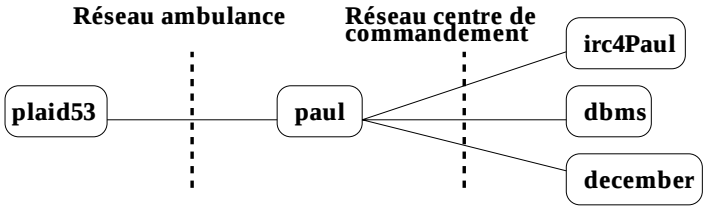

FIG. 9.1 – Répartition des composants du service d'urgence

La section suivante souligne, à travers des scénarios de déconnexion du service d'urgence médicale, les besoins en terme d'adaptation des applications nomades.

9.1.2 Assemblages et déconnexion : les besoins

Scénarios de déconnexion

Supposons que le PDA du secouriste *Paul* soit hors de portée de l'ambulance, dans ce cas, la connexion entre le *plaid53* et *Paul* est coupée. Une façon de gérer cette déconnexion consiste alors à remplacer le composant représentant une couverture *plaid53* par un composant représentant un éditeur `doctorKit` (le composant est local au PDA) permettant au secouriste d'entrer lui-même les données correspondant aux valeurs des constantes de vie. Dans la suite, ce scénario de déconnexion est référencé sous le nom « *sc-remplacer* ».

Cette fois, supposons que le secouriste soit hors de portée du réseau du centre de commandement. Dans ce cas, les connexions entre le composant de chat et *Paul*, le composant de base de données et *Paul* et le composant de gestion de l'historique et *Paul* sont coupées. Le composant de chat est vital pour le fonctionnement de la fonctionnalité `diagnose` du secouriste. Il est remplacé par le composant système expert (local au PDA) supportant la même fonction. D'autre part, la fonctionnalité `completeReport` est basculée en mode asynchrone. Le secouriste peut alors continuer à utiliser cette fonctionnalité tout en étant déconnecté et les rapports ne seront effectivement clôturés que lors d'une reconnexion avec le centre de commandement (à partir des données stockées via la communication asynchrone). Dans la suite, ce scénario de déconnexion est référencé sous le nom « *sc-retarder* ».

Considérons comme dernier cas une variante du scénario *sc-retarder* où le secouriste doit d'abord s'authentifier auprès du centre de commandement avant de pouvoir clôturer un rapport. En cas de déconnexion, l'authentification ne peut pas avoir lieu. Comme la fonctionnalité `completeReport` n'est pas vitale ni pour le travail du secouriste ni pour le patient, la politique choisie consiste à passer en mode dégradé pour la fonctionnalité `completeReport`. Ainsi, toutes les opérations de clôture de dossier sont interdites pendant la durée de déconnexion. Dans la suite, ce scénario de déconnexion est référencé sous le nom « *sc-dégrader* ».

Caractéristiques des composants liées à la mobilité

Jusque ici, nous avons vu que les adaptations d'applications « classiques » n'ont pas besoin d'être anticipées. C'est à dire que nous n'avons pas besoin de prévoir quelles adaptations peuvent avoir lieu pour concevoir les applications. Prendre en considération les applications mobiles implique de se positionner dans un contexte tout autre puisque dès la conception d'applications, il est nécessaire de prévoir comment celle-ci peut fonctionner en présence des déconnexions. En effet, les exemples de scénarios de la section précédente montrent qu'il n'existe pas une unique politique de gestion des déconnexions et que le choix d'une politique repose sur des caractéristiques architecturales d'une application mobile. Par exemple, nous avons vu que selon que l'on

utilise un authentificateur ou non, la politique de gestion de la déconnexion du réseau du centre de commandement n'est pas la même (passage en asynchrone ou en mode dégradé).

De ces scénarios, nous avons dégagé deux caractéristiques liées à la nomadicité qui nous paraissent essentielles pour gérer les déconnexions et qui doivent être spécifiées dès la conception de l'application pour la préparer aux déconnexions :

– le **degré de nécessité** d'une fonctionnalité ou d'un composant requis
– le **mode de communication** supporté par les composants

Par exemple, il est vital qu'un sauveteur puisse toujours effectuer un diagnostic, par contre, il n'est pas vital que le secouriste puisse clôturer à tout moment un rapport médical. `diagnose` est donc une fonctionnalité obligatoire alors que `completeReport` peut être optionnelle. Lorsque le sauveteur clôture un rapport, il n'a pas besoin d'être toujours connecté au composant d'historique pour effectuer sa tache : l'historique est un composant requis optionnel. Enfin, si le sauveteur a besoin d'être authentifié avant de pouvoir clôturer un rapport alors l'authentificateur est un composant requis obligatoire.

Notons que la prise en compte de caractéristiques liées à la nomadicité a également été proposé par [63]. Les auteurs introduisent deux méta-données appelées *déconnectabilité* et *nécessité* pour la conception d'applications mobiles. Un composant déconnectable est un composant dont certains composants requis peuvent être déconnectés lors d'un passage à un mode déconnecté. Un composant nécessaire est un composant dont la présence est obligatoire pour le fonctionnement en mode déconnecté. La méta-donnée de nécessité correspond au degré de nécessité. Quant à la méta-donnée de déconnectabilité, elle peut être déduite du mode de communication et du degré de nécessité. Une principale différence avec ces travaux est que les méta-données sont spécifiées statiquement alors que nos caractéristiques peuvent être modifiées pendant l'exécution.

Types d'adaptation élémentaire liées à la mobilité

Les scénarios de déconnexion correspondent à de nouveaux types d'adaptations élémentaires qui doivent être pris en charge afin de gérer les déconnexions de composants. Ainsi nous pouvons identifier plusieurs types d'adaptations élémentaires liées à la mobilité :

- Une **adaptation de remplacement** demande le remplacement d'un composant par un autre au sein d'un assemblage. Cette adaptation a lieu lors d'une déconnexion, quand un composant local remplace un composant auquel on est plus connecté. Elle a également lieu lors d'une reconnexion, quand on se connecte à un composant distant qui remplace alors le composant local qui était utilisé pendant la durée de déconnexion.

- Une **adaptation de communication** demande une modification du mode de communication effectif pour un ensemble de connexions. Cette adaptation est utilisée pour synchroniser des opérations ou au contraire pour les différer.

- Une **adaptation de nécessité** demande une modification du degré de nécessité d'un port (fourni ou émis). Cette adaptation est utilisée pour dégrader les fonctionnalités d'une application lorsque celle-ci ne dispose pas de toutes les ressources qu'elle requière. Et inversement, elle permet de revenir à un mode complet lorsque l'application dispose à nouveau de toutes les ressources nécessaires à son fonctionnement.

- Une **adaptation de déconnexion** demande une suppression d'un ensemble de connexions (un ou plusieurs ports émis deviendront manquants). Cette adaptation indique lorsqu'une coupure a lieu. C'est la seule adaptation qui a toujours lieu lors d'une déconnexion volontaire ou non.

9.1.3 Plates-formes à composants pour usagers mobiles

La programmation par composants permet de construire une application par assemblage de composants puis de modifier ou de remplacer dynamiquement certaines parties de l'application (retirer, ajouter un composant) avec un minimum d'interférences sur le reste du système. Les plates-formes à composants actuelles [89, 86, 79, 101, 17, 97] offrent ainsi des mécanismes permettant de faire évoluer l'application pour l'adapter aux besoins des utilisateurs ou aux conditions changeantes de l'environnement. Actuellement, il n'existe pas de solution partant de la conception jusqu'à l'exécution des applications ubiquitaires sous forme de composants. Cependant, des solutions à base de composants émergent pour proposer des plates-formes facilitant la découverte de service ou encore les déconnexions dans le cadre d'applications nomades car les approches à composants donnent des solutions techniques de mise en œuvre acceptables comme l'atteste les récentes extensions des plates-formes à composants OpenCCM [72] ou Fractal/Julia [21] par exemple.

Découverte de service et déploiement à la demande. Ces travaux [41] abordent la problématique de la découverte de services par les usagers mobiles. les services sont considérés comme des assemblages de composants dont une partie reste sur support fixe et dont l'autre partie est découverte, téléchargée et déployée sur des supports mobiles à la demande en fonction de la localisation et des possibilités chaque terminal. Une infrastructure a été implémentée au dessus de la plate-forme à composants OpenCCM. Les problèmes liés à la gestion des déconnexions n'y sont pas traités.

Gestion de la duplication et de la mise en cohérence. Domint [62] est une architecture orientée composants pour la gestion des déconnexions. Cette architecture est basée sur l'utilisation de conteneurs comme structure d'accueil pour intégrer un service de gestion des déconnexions. Domint repose sur la conception d'applications mobiles avec partitionnement des composants de l'application en composants déconnectables et composants non déconnectables [63]. La continuité de service est assurée par la création des composants déconnectés (copie de composants déconnectables) dans le terminal mobile qui journalisent les opérations exécutées localement en mode partiellement connecté ou déconnecté. Lors des reconnexions, les composants déconnectés et les composants distants sont réconciliés avec gestion de cohérence entre les différentes répliques du composant. Ce projet ne discute pas du cas où le composant invoqué est non déconnectable. Un prototype a été réalisé au-dessus de la plate-forme à composants OpenCCM [72].

D'autres travaux [70] utilisent le même principe. Les traitements garantissant la disponibilité des composants sont implémentés comme des extensions JavaPod[1]. Cette extension vise à introduire des mécanismes de duplication et de mise en cohérence pour assurer la disponibilité des applications en mode déconnecté. Les déconnexions traitées sont explicites et initiées par les usagers mobiles.

Dans cette approche, la duplication des composants localement au terminal mobile peut être considérée comme une **adaptation de remplacement**. Quant à la journalisation des opérations effectuées localement, elle peut être assimilée à une **adaptation de communication**.

Prise en compte du contexte d'exécution. JASON (Java Ad hoc Services on ad hOc Networks) [106] est une plate-forme à composants visant à offrir aux utilisateurs de terminaux mobiles une certaine continuité d'accès aux services proposés par des réseaux d'infrastructure. Cette approche repose sur une prise en compte du contexte d'exécution, sur un déploiement dynamique de services applicatifs sur les terminaux des usagers et sur une collaboration spontanée des terminaux en situation de mobilité. La plate-forme JASON est mise en œuvre

[1]Les composants de base fournis par JavaPod (serveurs, conteneurs, talons et squelettes) sont génériques et ne contiennent aucun traitement en ce qui concerne les propriétés non fonctionnelles. Le comportement des quatre types de composants de la plate-forme peuvent être étendus à l'aide d'entités, appelées extensions. La composition des extensions pour une entité donnée permet l'intégration de certaines propriétés non fonctionnelles.

par un ensemble de services OSGi [93]. Ces services OSGi sont conçus pour s'adapter dynamiquement en fonction des conditions d'exécution afin déployer sur les terminaux des utilisateurs des composants rendant des services utilisateurs similaires à ceux proposés par les composants distants accédés en mode connecté aux réseaux d'infrastructures. Ici encore, l'utilisation en local de composants «similaires» aux composants distants correspond à une **adaptation de remplacement**.

Gestion d'un mode dégradé. Les travaux de [51] proposent d'exploiter un modèle de composants hiérarchiques afin de répercuter les déconnexions des équipements sur l'architecture du composant en rendant inactives certaines de ses interfaces tout en autorisant un fonctionnement dégradé du composant. Une extension du modèle à composants Fractal permet de distribuer physiquement un composant composite sur plusieurs machines. Ce dernier exhibe ses interfaces sur un ensemble de machines même si ses sous-composants sont localisés. La gestion décentralisée de l'architecture du composite offre un support pour la prise en compte des déconnexions qui sont susceptibles d'entraîner des ruptures de liaisons entre sous-composa nts. Pour cela, la notion d'interface active est ajoutée au modèle hiérarchique afin d'isoler les dépendances entre interfaces requises et fournies. Les déconnexions induisent la désactivation de certaines interfaces qui se propage alors dans la hiérarchie de composants. Les interfaces restant actives autorisent un composant à continuer de fonctionner, bien que dans un mode dégradé. Le passage en mode actif ou inactif d'une interface peut être considéré comme une **adaptation de nécessité**.

9.2 Extension du modèle et des propriétés

Le modèle Satin permet de préserver la sûreté des assemblages de composants face aux adaptations dynamiques dans le cadre des plates-formes à composants traditionnelles. Aussi proposons-nous de considérer les déconnexions comme des adaptations d'assemblages de composants afin de voir si nos travaux autour de la sûreté des composants peuvent être réutilisés. En effet, lors d'une déconnexion, un composant peut devenir manquant car non accessible via le réseau et l'utilisateur peut décider soit de le remplacer par un composant local capable d'assurer la même fonction (similitude avec un remplacement de composant) soit de mettre en attente les requêtes vers ce composant jusqu'à reconnexion (similitude avec un retrait de composant). Des attentes plus relatives aux applications nomades telles que passer en asynchrone ou encore fonctionner en mode dégradé doivent également pouvoir être mises en œuvre comme des adaptations à contrôler.

La prise en compte d'un nouveau domaine, en l'occurrence la mobilité, entraîne la modélisation de nouvelles caractéristiques des composants et de nouveaux types d'adaptations élémentaires. De nouvelles propriétés visant à assurer la sûreté de fonctionnement des applications face à ces nouvelles adaptations sont donc définies. Les propriétés doivent alors être formalisées puis validées avec le reste des propriétés existantes. Nous proposons de définir ce nouveau modèle que nous appelons `SatinNexus` comme un raffinement du modèle abstrait `SatinBase` qui a été défini en 5.2. Dans la suite de ce chapitre, nous utilisons Picore pour décrire les différents raffinements de concepts.

9.2.1 Extension des propriétés de sûreté

Au départ, nous avons identifier 10 propriétés de sûreté liées au « Quoi » regroupées en 6 familles (cf chapitre 5). Les adaptations doivent préserver ces propriétés pour pouvoir être mises en œuvre. La prise en compte de la mobilité des composants au niveau du modèle impacte les propriétés de sûreté. Certaines sont modifiées et de nouvelles sont ajoutées.

Seules les propriétés P_{1b} et P_3 sont impactées par la prise en compte des caractéristiques liées à la mobilité (nécessité et mode de communication) :

- **P_{1b}** : **Conservation des fonctionnalités de base**. Ce ne sont plus toutes les fonctionnalités initiales d'un composant qui ne doivent pas être perdues au fil du temps mais seulement les fonctionnalités obligatoires.
- **P_3** : **Consistance des assemblages point à point**. Tout ce qui est utilisé par un composant dans le cadre d'une adaptation doit toujours être fourni par un autre composant (ou par lui-même). Cette propriété inclut à présent le fait que le mode de communication effectif soit supporté par les deux parties. Cela se traduit par une extension de la relation de conformité au sens du filtrage.

La prise en compte au niveau du modèle de la distinction entre mode de communication synchrone et asynchrone et entre les composants présents et les composants manquants au sein des assemblages permet de gérer deux nouvelles propriétés :

- **P_6** : **Deadlocks**. Les cycles et les points de non-déterminisme (bénins ou malins) sont interdits à l'intérieur d'assemblages synchrones lorsque des composants sont « manquants ». Cette propriété est importante pour éviter les inter blocages.
- **P_7** : **Homogénéité des communications**. Le mode de communication effectif à l'intérieur d'un flot d'exécution doit être unique. Cette propriété est importante pour éviter les appels bloquants.

Dans le chapitre 7, nous avons vu que les propriétés sont assurées par des contraintes sur les classes et opérations du modèle et sont décrites à l'aide du langage OCL [120]. Ces contraintes ont été modifiées pour prendre en compte la mobilité des composants et d'autres contraintes ont été ajoutées. L'ensemble doit être validé comme nous l'avons vu pour les contraintes de base.

9.2.2 Raffinement du modèle abstrait

Pour gérer les applications mobiles, il est nécessaire de raffiner le modèle abstrait pour prendre en compte les caractéristiques et les adaptations liées à ce type d'applications.

Prise en compte des caractéristiques de composants liées à la mobilité

Un rôle de composant permet de décrire ce que fournit et ce que requiert le composant ainsi que les adaptations qu'il a subies. Ainsi, un rôle a des ports fournis représentant les fonctionnalités du composant. Nous distinguons à présent deux types de port fourni, **obligatoire** ou **optionnel**, et trois modes de communication qu'un port fourni supporte, **synchrone**, **asynchrone** ou **mixte**. Un rôle comporte également des ports émis correspondant aux fonctionnalités des composants requis (utilisés par adaptation) par le composant. Comme pour les ports fournis, un port émis est soit obligatoire soit optionnel et porte le mode de communication effectif de la connexion (synchrone ou asynchrone).

Modélisation des adaptations à la déconnexion

Une déconnexion peut être traitée comme une adaptation particulière : une adaptation d'une connexion entre deux composants ou d'un assemblage au lieu d'une adaptation d'un ou plusieurs composants. Dans le chapitre 5, nous avons vu qu'à chaque fois qu'un schéma d'adaptation est appliqué à un ensemble de composants, une instance d'adaptation représentant un ensemble d'assemblages est créée. Les instances d'adaptation sont donc les entités manipulées lors des déconnexions.

Un **schémas de mobilité** est un ensemble d'**adaptations élémentaires** définies sur un ensemble de rôles de schémas d'adaptation et de rôles de composants (les paramètres du schéma de mobilité). Un schéma de mobilité s'applique donc à un ensemble d'instances d'adaptation et à un ensemble de composants.

Pour pouvoir déconnecter un ensemble d'assemblages de composants il faut pouvoir appliquer les adaptations élémentaires d'un schéma de mobilité à ces assemblages (c'est à dire à des instances d'adaptation). Cette opération a alors pour effet de modifier les assemblages et le rôle des composants (modification du mode de communication effectif, ...). Notons que contrairement à l'application d'un schéma d'adaptation classique qui modifie uniquement les rôles des composants participants directement à l'adaptation, l'application d'un schéma de mobilité peut impacter tous les composants (et donc leurs rôles) par propagation des changements effectuées sur les caractéristiques des composants liées à la mobilité.

A l'instar des rôles de composants qui sont déduits des adaptations élémentaires des schémas d'adaptation et qui spécifient des caractéristiques à satisfaire pour qu'un schéma d'adaptation s'applique (par exemple, le composant doit fournir un certain port), les rôles de schémas d'adaptation sont déduits des adaptations élémentaires des schémas de mobilité et spécifient des caractéristiques à satisfaire pour qu'un schéma de mobilité s'applique (par exemple, un schéma d'adaptation doit définir deux adaptations élémentaires).

A l'instar des schémas d'adaptation, les schémas de mobilité ne peuvent être appliqués que s'ils satisfont des propriétés de sûreté. Les propriétés à garantir en cas de déconnexion concernent alors aussi bien les problèmes spécifiques à la mobilité tels que le changement de communication ou la dégradation des fonctionnalités que les problèmes classiques tels que la consistance d'assemblage.

Description Picore

En utilisant Picore, la notion de port du modèle abstrait `SatinBase` est raffinée et un nouveau type de schéma est introduit de la manière suivante.

```
metamodel SatinNexus refines SatinBase
    class ProvidedPort is refined
        attribute isMandatory 1 boolean
        attribute asynchronEnabled 1 boolean
        attribute synchronEnabled 1 boolean

    class EmittedPort is refined
        attribute isMandatory 1 boolean
        attribute isSynchronConnection 1 boolean

    class MobilityPattern
        operation createFrom(adaptationRoles : collection,
                             componentRoles : collection,
                             adaptations : collection) : void
        operation instantiate(adaptationInstances : collection,
                              components : collection) : void
        ...

    class AdaptationRole
        attribute numberOfParticipants 1 int
        attribute numberOfElementaryAdaptation 1 int
```

9.2.3 Raffinement des modèles requis

Prise en compte des caractéristiques de composants liées à la mobilité

Le mode de communication supporté par les ports fournis d'un composant ainsi que leur degré de nécessité doivent être spécifiés à la conception et sont donc déduits de l'implantation des composants via le modèle requis d'extraction des données `ExtractionExtension` qui doit donc être raffiné.

Par contre, le mode de communication effectif des ports émis ainsi que leur degré de nécessité dépendent des assemblages formés par adaptation. Comme les adaptations élémentaires permettent de décrire des assemblages de composants, elles doivent permettre de spécifier le mode de communication entre les composants et définir le degré de nécessité des composants requis. Ainsi, le modèle requis d'adaptations élémentaires (`AdaptationExtension`) doit être raffiné pour enrichir l'introspection des adaptations.

Modélisation des adaptations à la déconnexion

Comme les schémas d'adaptation permettent de décrire l'adaptation de composant par assemblage, les schémas de mobilité permettent de décrire l'adaptation d'assemblages de composants (i.e. : instances d'adaptation) par politiques de déconnexion. Nous prenons en compte plusieurs types d'adaptations élémentaires liées à la mobilité : **adaptation de remplacement**, **adaptation de communication**, **adaptation de nécessité** et **adaptation de déconnexion**. Il faut raffiner le modèle d'adaptations élémentaires pour prendre en compte ces nouveaux types d'adaptation.

Description Picore

La description Picore suivante décrit le raffinement des modèles requis `ExtractionExtension` et `AdaptationExtension`.

```
metamodel ExtractionExtensionRaffinement refines ExtractionExtension
    class Feature is refined
        operation getCommunicationMode() : String
        operation isMandatory() : boolean

metamodel AdaptationExtensionRaffinement refines AdaptationExtension
    class ElementaryAdaptation is refined
        operation getType() : String in {'add', 'control',
                                         'replace', 'communication',
                                         'contingency', 'cut'}
        operation isSynchronConnection() : boolean
        operation getMandatoryComponentRole() : collection
        operation getOptionnalComponentRole() : collection
```

9.2.4 Raffinement du modèle d'adaptations élémentaires `ASLExtension`

Rappelons que nous fournissons un modèle d'adaptations élémentaires, `ASLExtension`, pour les plates-formes ne proposant pas de modèle d'adaptations élémentaires compatible avec le modèle requis `AdaptationExtension`. Ce modèle doit être raffiné pour rester conforme à `AdaptationExtension` qui a été raffiné pour gérer les déconnexions.

Prise en compte des caractéristiques de composants liées à la mobilité

La grammaire du langage ASL, décrite dans le chapitre 6 et dans l'annexe A, a été étendue pour prendre en compte les nouveaux éléments de modélisation des adaptations. La règle de production `adaptation` est modifiée comme suit. La règle de production `necessity` permet de désigner les rôles de composants obligatoires et optionnels). Si elle n'est pas utilisée, tous les rôles requis sont obligatoires par défaut. L'utilisation de l'opérateur "=>" signifie que le mode de communication entre les composants de l'adaptation est asynchrone ("->" pour une communication synchrone).

```
adaptation := "newRole" typeName "on" ident "->" methodDeclarations
            | "newPort" [ "!" ] methodDeclaration [ necessity ] ("=>" | "->") exp
            | "modifyPort" [ "!" ] methodDeclaration [ necessity ] ("=>" | "->") exp
necessity := "mandatoryRoles{" necessityList "}" "optionalRoles{" necessityList "}"
           | "mandatoryRoles{" necessityList "}"
           | "optionalRoles{" necessityList "}"
necessityList := ident { "," ident }
```

Modélisation des adaptations à la déconnexion

Pour prendre en compte la description des adaptations élémentaires des schémas de mobilité, la grammaire du langage ASL a été à nouveau étendue. La règle de production `root` est modifiée comme suit. La règle de production `mobilityAdaptation` permet de définir les quatre types d'adaptations élémentaires des schémas de mobilité explicitées un peu plus haut.

```
root := "adaptationPattern" ident "(" formalParameters ")" "{" adaptations "}"
      | "mobilityPattern" ident "(" formalParameters ")" "{" mobilityAdaptations "}"

mobilityAdaptations := mobilityAdaptation { "," mobilityAdaptation }

mobilityAdaptation := "replace" patternParam "=" ident
                    | "communication" patternElementaryAdaptation "=" ( "synchron" | "asynchron" )
                    | "necessity" patternParam
                    | "disconnect" patternElementaryAdaptation

patternParam := ident ".param" access

patternElementaryAdaptation := ident ".adp" access

access := "[" [ "1"-"9" ] "0"-"9" { "0"-"9" } "]"
```

Par exemple, le schéma de mobilité `replaceComponent` décrit ci-après autorise, s'il est applicable, de remplacer un composant dans un assemblage défini par un schéma d'adaptation. Ce schéma peut potentiellement s'appliquer à toute instance d'adaptation issue d'un schéma comprenant au moins deux paramètres.

```
1 mobilityPattern replaceComponent (pattern, newComponent) {
2     replace pattern.param[2] = newComponent
3 }
```

Le schéma de mobilité `cutConnection` décrit ci-après autorise, lorsqu'il est applicable, simplement de couper toutes les connexions à l'intérieur de l'assemblage défini par la première adaptation élémentaire d'un schéma d'adaptation. Ce schéma peut potentiellement s'appliquer à toute instance d'adaptation issue d'un schéma comprenant au moins une adaptation élémentaire.

```
1  mobilityPattern cutConnection (pattern) {
2     disconnect pattern.adp[1]
3 }
```

Enfin, le schéma de mobilité `deferConnection` décrit ci-après fait la même chose que le précédent et autorise de passer en asynchrone toutes les communications à l'intérieur de l'assemblage défini par la première adaptation élémentaire d'un schéma d'adaptation. Ainsi, même si toutes les connexions à l'intérieur de cet assemblage sont coupées, les requêtes seront transmises plus tard lors d'une reconnexion. Ce schéma peut s'appliquer potentiellement à toute instance d'adaptation issue d'un schéma comprenant au moins une adaptation élémentaire.

```
1 mobilityPattern deferConnection (pattern) {
2     disconnect pattern.adp[1] ,
3     communication pattern.adp[1] = asynchron
4 }
```

Description Picore

La description Picore suivante décrit le raffinement de `ASLExtension`.

```
metamodel ASLExtensionRaffinement refines ASLExtension
    class AdaptationRule is refined
        attribute isSynchronConnection : boolean
        attribute mandatoryComponentRoleList : collection
        attribute optionnalComponentRoleList : collection
        operation isSynchronConnection() : boolean
        operation getMandatoryComponentRole() : collection
        operation getOptionnalComponentRole() : collection

    class ReplaceRule extends AdaptationRule
    class CommunicationRule extends AdaptationRule
    class NecessityRule extends AdaptationRule
    class DeconnectionRule extends AdaptationRule
```

9.3 Modélisation du service d'urgence en Satin

Considérons la modélisation en Satin de l'exemple fil rouge décrit en 9.1.1.

Conception de l'application en tenant compte des caractéristiques liées à la nomadicité

Le rôle `EmergencyAssistant` d'un composant représentant le sauveteur *Paul* comporte les ports fournis `setTemperature`, `setCardiacPace`, `setRespiratoryRythm`, `diagnose` et `completeReport`. La figure 9.2 décrit le rôle associé au composant représentant le sauveteur *Paul* et met en évidence les caractéristiques spécifiques à la mobilité des ports fournis. Il est possible d'interagir avec *Paul* de manière synchrone ou asynchrone pour lui communiquer les constantes vitales de ses patients ainsi que pour clôturer un dossier médical. Par contre, déterminer un diagnostic ne peut être différé. Enfin, seule la clôture d'un dossier peut être une tâche indisponible momentanément : le sauveteur doit pouvoir faire un diagnostic et consulter les constantes vitales d'un patient à tout moment.

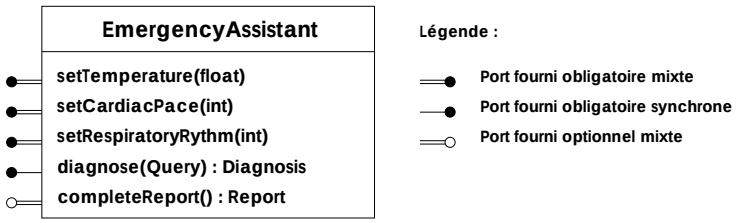

FIG. 9.2 – Rôle associé à *Paul*

Description des assemblages de composants par définition de schémas d'adaptation

Pour décrire les assemblages des composants de l'exemple fil rouge, il est nécessaire de définir des schémas d'adaptation. Considérons les quatre schémas d'adaptation `VitalConstantRecording`, `MedicalExpertise`, `PersistenceAndStat` et `AuthentificationPersistenceAndStat` décrits ci-dessous à l'aide du langage ASL étendu. Les trois premiers décrivent respectivement une adaptation qui permet le transfert des données d'un composant jouant le rôle de la gauge qui prend des mesures (`Gauge`) vers un composant jouant le rôle du secouriste (`Rescuer`), l'aide au diagnostic, et le suivi et la persistance des données médicales constituant les dossiers des patients. Le dernier schéma est utilisé à la place de `PersistenceAndStat` dans le scénario *sc-dégrader* défini en 9.1.1.

```
1  adaptationPattern VitalConstantRecording (Rescuer rescuer, Gauge gauge) {
2    modifyPort gauge.sendData(float temperature, int cardiacPace, int respiratoryRythm)
3    optionalRoles{rescuer} => rescuer.setTemperature(temperature);
4                              rescuer.setCardiacPace(cardiacPace);
5                              rescuer.setRespiratoryRythm(respiratoryRythm)
6 }
7
8  adaptationPattern MedicalExpertise (Rescuer rescuer, Expert expert) {
9     modifyPort rescuer.diagnose(Query q) : Diagnosis
10    mandatoryRoles{expert} -> Diagnosis d = expert.diagnose(q)
11 }
12
13 adaptationPattern PersistenceAndStat (Rescuer rescuer, Storage storage, Statistic stat) {
14    modifyPort rescuer.completeReport() : Report
15    optionalRoles{storage, stat} -> Report r = rescuer._call();
16                                    storage.store(r);
17                                    stat.addEvent(r)
18 }
19
21 adaptationPattern
22   AuthentificationPersistenceAndStat (Rescuer rescuer, Authentificator auth,
23                                       Storage storage, Statistic stat) {
24    modifyPort rescuer.completeReport() : Report
25    mandatoryRoles{auth} optionalRoles{storage, stat} -> Report r = rescuer._call();
26                                                         auth.check();
27                                                         storage.store(r);
28                                                         stat.addEvent(r)
29 }
```

Dans le premier schéma, `Gauge` est un rôle déduit de l'unique adaptation élémentaire du schéma et fournit le port `sendData`. De la même manière, le rôle déduit `Rescuer` fournit les ports `setTemperature`, `setCardiacPace` et `setRespiratoryRythm`. Le mode de communication entre la gauge et le secouriste est asynchrone et `Rescuer` est un rôle de composant requis optionnel pour `Gauge`.

Assemblage des composants par application de schémas d'adaptation

La figure 9.3 présente l'architecture de l'application après acceptation de l'application des trois schémas d'adaptation `VitalConstantRecording`, `MedicalExpertise` et `PersistenceAndStat` aux composants concernés. Par exemple, l'assemblage formé par *Paul* et *Plaid53* est obtenu en appliquant le schéma d'adaptation `VitalConstantRecording` à *Plaid53* (en tant que gauge) et *Paul* (en tant que secouriste).

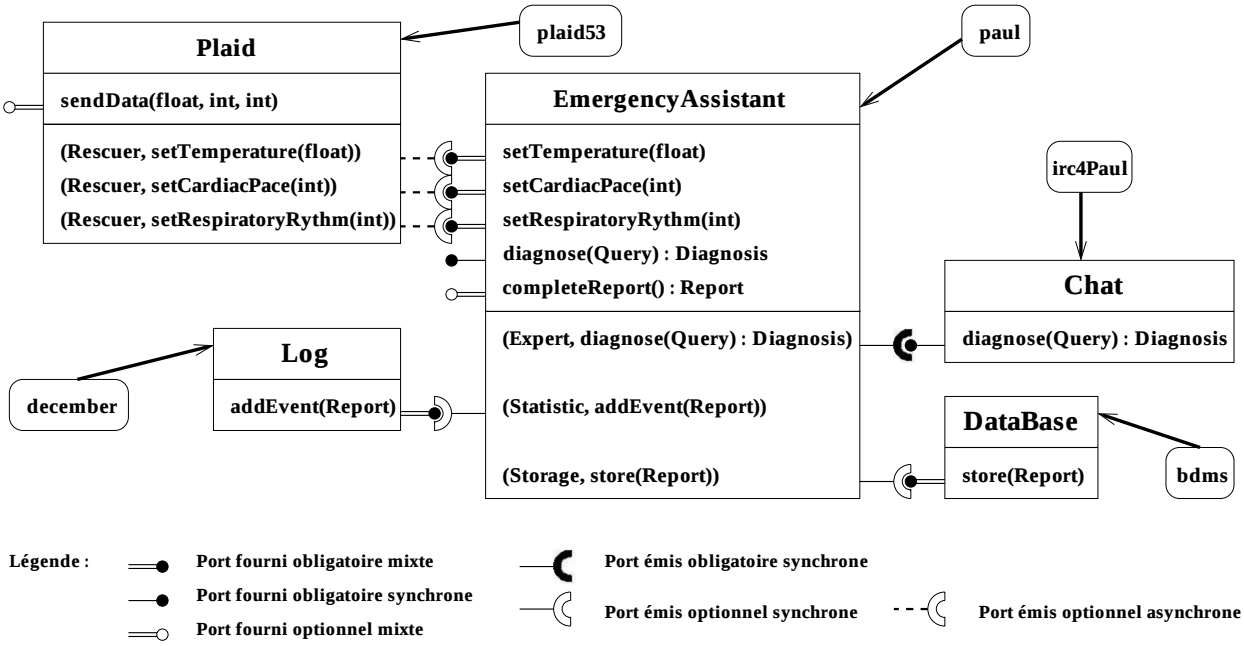

FIG. 9.3 – Evolution des rôles

La figure 9.4 décrit les instances d'adaptation liant les différents composants de l'application fil rouge. Ainsi, l'instance d'adaptation résultant de l'application du schéma d'adaptation `VitalConstantRecording` à *Plaid53* et *Paul* est *VitalConstantRecording1*.

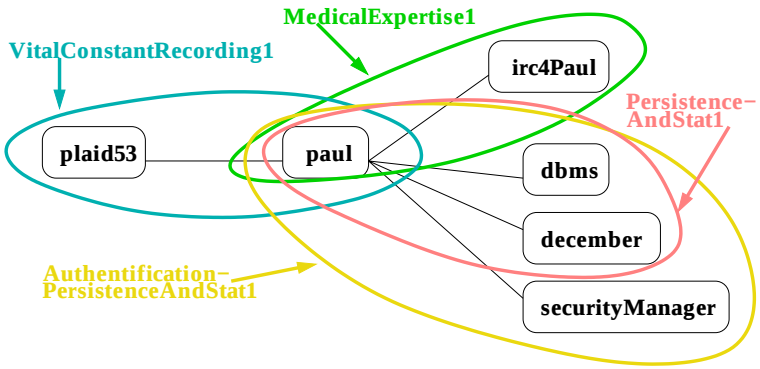

FIG. 9.4 – Instances d'adaptation liant *Paul*, *plaid53*, *irc4Paul*, *dbms*, *december* et *securityManager*

Gestion des déconnexions par application de schémas de mobilité

Si l'on reprend les exemples de scénarios présentés en 9.1.2, la politique de gestion de déconnexion du scénario *sc-remplacer* est mise en place en appliquant :

1. le schéma de mobilité `replaceComponent` à l'instance d'adaptation *VitalConstantRecording1* et au composant de remplacement *doctorKit* représentant un éditeur[2]

Notons que le schéma de mobilité ne peut être appliqué que dans la mesure où *doctorKit* est substituable à *plaid53* sinon la propriété $\mathbf{P_3}$ est violée.

De même, la politique de gestion de déconnexion du scénario *sc-retarder* est mise en place en appliquant :

1. le schéma de mobilité `replaceComponent` à l'instance d'adaptation *MedicalExpertise1* et au composant de remplacement *helpMeDoctor* représentant un système expert
2. le schéma de mobilité `deferConnection` à l'instance d'adaptation *PersistenceAndStat1*

Notons que si la base de données ou l'historique ne supportait pas les communications asynchrones, le schéma de mobilité `deferConnection` n'aurait pas pu être appliqué dans le scénario *sc-retarder* sans violer la propriété $\mathbf{P_3}$ et Satin aurait refusé cette politique de déconnexion. Dans ce cas, une alternative sûre consisterait à utiliser le schéma de mobilité `cutConnection` comme dans le scénario *sc-dégradé* défini dans 9.1 pour gérer la déconnexion qui cette fois serait validée par Satin.

Enfin, la politique de gestion de déconnexion du scénario *sc-dégrader* est mise en place en appliquant :

1. le schéma de mobilité `replaceComponent` à l'instance d'adaptation *MedicalExpertise1* et au composant de remplacement *helpMeDoctor* représentant un système expert
2. le schéma de mobilité `cutConnection` à l'instance d'adaptation *AuthentificationPersistenceAndStat1* (le schéma de mobilité `deferConnection` n'est pas utilisable puisque l'authentificateur ne fonctionne qu'en mode synchrone)

Notons que si le port fourni `completeReport` avait été obligatoire, le schéma de mobilité `cutConnection` n'aurait pas pu être appliqué dans le scénario *sc-dégradé*. En effet, un port fourni obligatoire aurait été manquant et la propriété de sûreté $\mathbf{P_{1b}}$ aurait été violée. Dans ce cas, une alternative consiste à laisser les communications avec l'authentificateur, la base de données et l'historique en synchrone mais au risque de compromettre la propriété $\mathbf{P_6}$ si un appel à `completeReport` se trouve dans un cycle ou est à l'origine d'un point de non déterminisme.

9.4 Conclusion

Extensibilité du modèle

Cette étude pour la prise en compte des applications mobiles nous a permis de valider l'extensibilité du modèle Satin vis-à-vis des adaptations de composants sans une remise en cause de la méthode et du fonctionnement de base du modèle. En particulier, la notion de schéma d'adaptation permet d'exprimer facilement les assemblages. Notons cependant qu'au niveau de ces schémas, la notion de mode de communication synchrone, asynchrone et mixte devient importante

[2] Il est donc nécessaire d'avoir le composant de remplacement avant la déconnexion.

lorsqu'on s'intéresse à des assemblages de composants réalisant une application nomade alors qu'elle n'influe que rarement sur l'adaptation d'une application « traditionnelle ». De même, la possibilité d'exprimer les composants obligatoires ou optionnels au niveau de ces schémas est essentielle pour pouvoir déterminer le degré de sûreté de certaines politiques de déconnexion telles que le fonctionnement en mode dégradé. Enfin, il s'est avéré naturel que les déconnexions s'expriment au dessus des assemblages comme des adaptations de schémas d'adaptations car elles influent essentiellement sur la façon dont sont construits les assemblages.

Les propriétés spécifiques à la déconnexion sont peu nombreuses, elles sont en général une simple extension des propriétés existantes. Seuls les moments de vérifications diffèrent, certaines propriétés étant vérifiées à la création, à l'application ou au retrait de schémas d'adaptation, c'est à dire avant une adaptation classique, et d'autres étant vérifiées à l'application de schémas de mobilité, c'est à dire avant une déconnexion lorsque celle-ci est volontaire.

Perspectives en ce qui concerne la gestion de la mobilité

Dans les cas où la déconnexion est involontaire, l'application d'un schéma de mobilité destiné à pallier aux problèmes dus à la coupure ne peut pas être anticipée. Cependant, pour conserver l'avantage de l'anticipation des erreurs liées à une adaptation, il semble intéressant d'envisager, dans le cas particulier de la gestion des déconnexions, d'utiliser le service de sûreté par avance, pour simuler les différents cas de figures qui peuvent se présenter lors d'une déconnexion pour une application donnée.

Lors de la reconnexion, on doit pouvoir se reconnecter à un composant différent (c'est le cas quand on se reconnecte à un réseau différent après un déplacement par exemple) mais compatible par rapport aux besoins en terme d'assemblage. Actuellement, le problème de la découverte de service n'est pas traité : comment trouver sur le nouveau réseau un composant susceptible de remplacer le « composant déconnecté » ? L'idée est de laisser à la plate-forme visée la gestion de la découverte de service par la modélisation d'un nouveau modèle requis.

De plus, il faudrait également vérifier que le nouveau composant soit dans un état (les données) adéquat pour son utilisation. Ce problème se rencontre également lorsqu'on cherche à remplacer un composant par un autre ou encore lors d'un changement de version d'un composant (changement d'implantation). Nous n'abordons pas le problème du transfert d'état actuellement. La gestion du transfert d'état impliquerait de « remonter », au niveau des rôles, les instances complètes (données comprises) et de définir de nouvelles propriétés de sûreté.

Enfin, pour plus de précision dans les vérifications effectuées, il est nécessaire de prendre en considération le problème de la détection de la déconnexion [116]. En effet, certains appels peuvent être émis vers des entités qui ne sont plus disponibles entre le moment où la déconnexion a lieu et le moment où elle est effectivement détectée. Ces appels posent alors problème car ils ne peuvent pas être traités par la cible (l'entité étant déconnectée) et ne sont pas non plus pris en compte par la politique de gestion des déconnexions qui va être appliquée suite à la détection. Une perspective consisterait à monitorer les appels au niveau du modèle.

« Comment la fin justifierait-elle les moyens ? Il n'y a pas de fin, seulement des moyens à perpétuité. »

René Char

« Un discours de fin de dîner devrait être comme la robe d'une jolie femme : assez long pour couvrir le sujet et assez court pour rester intéressant. »

Ralph Emerson

10 Conclusion et perspectives

LES canevas orientés aspects et les modèles de composants sont des approches qui tendent à se généraliser pour permettre l'adaptation dynamique des applications. Dans la première partie de ce document nous avons discuté des points forts et des points faibles en terme de préservation de la sûreté de fonctionnement des applications par ces approches. Nous avons souligné comme possibilités réduites de ces propositions technologiques les suivantes :

Limite 1 Les types d'erreurs induits par des adaptations cachées ne sont pas pris en charge.

Limite 2 Les différences de mise en œuvre des adaptations marquent des différences en terme de vérifications effectuées : les solutions sont donc spécifiques aux besoins de chaque plate-forme et ne peuvent pas être réutilisées dans d'autres plates-formes.

Limite 3 Les vérifications effectuées ne sont pas systématiques et les techniques de vérification employées sont souvent à la charge du développeur, intégrateur ou adaptateur ;

Limite 4 Les erreurs ne sont pas suffisamment anticipées ;

Limite 5 Les techniques utilisées pour déterminer quand il est sûr d'adapter les composants présentent des inconvénients majeurs en terme de performance ou de flexibilité.

La motivation de notre travail a été de fournir une réponse à ces limitess en proposant une approche pouvant être utilisée dans plusieurs plates-formes mais capable de bien répondre aux spécificités de chaque plate-forme. Notre solution a eu pour objectifs de :

Objectif 1 Abstraire le concept d'adaptation pour fournir une solution indépendante des mises en œuvre des adaptations dans les plates-formes existantes ;

Objectif 2 Etendre la notion de type pour prendre en compte la gestion des évolutions structurelles et comportementales des entités logicielles adaptées dynamiquement ;

Objectif 3 Etablir une méthodologie pour formaliser et valider notre solution.

Dans un premier temps, nous résumons les propositions discutées dans ce document ainsi que leurs apports. Nous discutons ensuite des perspectives qui ont émergées de nos travaux et expérimentations.

10.1 Evaluation des travaux réalisés

Notre solution pour empêcher l'introduction de fautes lors d'adaptations dynamiques a permis d'atteindre nos trois principaux objectifs. En effet, le modèle abstrait Satin permet de définir des propriétés de sûreté que les adaptations doivent préserver sans considérer les aspects « technologiques » de ces adaptations, ce qui permet d'atteindre le premier objectif.

La notion de rôle introduite par le modèle Satin offre une nouvelle forme de typage dans laquelle sont pris en compte les aspects dynamiques des entités logicielles. Les rôles permettent l'évolution de type à l'exécution afin de réaliser le second objectif.

Le troisième objectif a été pris en compte à travers l'utilisation des formalismes OCL [120] pour décrire les propriétés de sûreté et B [1] pour les valider.

La suite de cette section montre comment notre solution répond aux limites de l'état de l'art citées ci-dessus et met en évidence les points forts de notre proposition.

Prise en compte des limites présentées dans l'état de l'art

La mise en évidence de types d'adaptations élémentaires et de leur impact entre eux répond à la première limites en faisant ressortir clairement les adaptations cachées.

La définition d'un modèle de sûreté d'adaptations indépendamment des plates-formes techniques répond à la seconde limites en fournissant des propriétés de sûreté réutilisables pour chaque plate-forme à travers la mise en œuvre d'un service de sûreté. Les plates-formes peuvent interroger le service afin de déterminer en fonction d'une adaptation donnée si celle-ci peut être effectuée sans compromettre la sûreté de fonctionnement de l'application à adapter.

La classification à la fois des approches étudiées dans l'état de l'art et des propriétés de sûreté vis-à-vis de ces types d'adaptations élémentaire nous permet de connaître exactement quel sous-ensemble des propriétés de sûreté doit être vérifié pour chaque plate-forme. Ceci permet de pallier à la troisième limites dans le sens où la vérification des propriétés de sûreté est systématique et où la mise en œuvre de cette vérification via le service de sûreté la rend transparente vis-à-vis des différents acteurs du processus logiciel.

La vérification des propriétés de sûreté est faite a priori par l'utilisation de contraintes OCL qui assurent qu'une opération d'adaptation ne peut être réalisée si elle ne maintient pas les propriétés de sûreté. Ainsi, les erreurs qui peuvent être introduites par une adaptation sont détectées avant même que l'adaptation soit effectuée. Ce point répond à la quatrième limites.

La dernière limites est contournée par la définition d'un protocole d'application associé à l'utilisation du service de sûreté. La prise en compte du moment d'adaptation correspond donc à un ordonnancement du processus d'adaptation à travers l'utilisation du protocole d'application du service et est également fonction du type des adaptations élémentaires visées. Ceci est un atout en terme de flexibilité.

Points forts de l'approche

La définition d'un ensemble de propriétés de sûreté « arbitraire » ne signifie pas comme nous l'avons mentionné plus haut que toutes les propriétés doivent être effectuées pour toutes les plates-formes. Les propriétés ont été définies de manière orthogonale de sorte que la vérification de l'une n'a pas d'impact sur la vérification des autres propriétés. **Ainsi, bien que générique, la solution est également du « sur mesure » à travers la vérification d'un sous-ensemble des propriétés de sûreté approprié à chaque plate-forme**. Ceci est réalisé en pratique par la configuration du service de sûreté.

Même si la vérification liée à certaines propriétés de sûreté est déjà effectuée par la plate-forme, la délégation de cette vérification peut être un point fort pour la plate-forme. Par exemple, la plupart des modèles à composants, tels que CCM [89] ou Fractal [86], prennent en charge la vérification de type au sein des assemblages. Actuellement, le prix à payer pour les plates-formes basées sur ces modèles est un manque de souplesse au niveau des adaptations qu'elles permettent : le type des composants dans ces plates-formes ne peut être modifié à l'exécution. **En déléguant cette vérification au service de sûreté ces plates-formes à composants peuvent s'affranchir de cette limite.**

Le découpage du modèle Satin en un modèle fourni représentant le cœur de l'approche et des modèles requis représentant les dépendances vers les plates-formes technologiques permet la séparation des préoccupations dès la conception. L'univers n'est plus seulement modélisé dans un but particulier : **le modèle précise également les conditions sous lesquelles il est utilisable.**

La démarche de formalisation des propriétés de sûreté au niveau du modèle abstrait permet également de les valider à ce même niveau. L'utilisation de la méthode B est un atout pour la vérification de modèles. **Le découpage en modèles fournis et requis facilite la vérification en séparant les éléments qui doivent être prouvés des éléments utilisables dans la preuve.**

L'approche puzzle est un support pour la transformation des modèles exécutables. Les modèles requis servent dans ce cas à décrire les points d'accroche du modèle dans les plates-formes. Le modèle est ensuite concrétisé sous forme d'un service que les plates-formes peuvent interroger. **La transformation de modèles par concrétisation d'un service pour plates-formes rend le processus de transformation indépendant de la cible.**

La formalisation des propriétés en OCL et la concrétisation du modèle Satin sous la forme du service de sûreté avec interprétation de l'OCL évite d'avoir à revalider les propriétés de sûreté après transformation. **L'approche service offre donc la transparence des vérifications effectuées en terme de sûreté aussi bien que la validation de ces vérifications.**

Pour conclure, l'approche puzzle couplée à la mise à disposition du modèle exécutable sous forme de service répond au RFP de l'OMG autour des modèles UML exécutables en fournissant un support à la construction, à la vérification, à la transformation et à l'exécution de modèles exécutables.

10.2 Perspectives

Poursuite du travail dans le domaine de la mobilité

Ce point est certainement la perspective de travail à plus court terme. Nous avons étudié un certain nombre de problèmes liés à la déconnexion de composants. Les propriétés de sûreté ont été étendues et de nouvelles ont été ajoutées. Il reste à valider l'ensemble par les mêmes principes que précédemment : simulation et méthode de preuves formelles.

D'autre part, l'aspect déconnexion n'est qu'un point lié à la mobilité parmi d'autres. En particulier, ne modélisant pas le processus d'adaptation même, nous n'avons pas distingué une adaptation d'une auto-adaptation. Dans le contexte de la mobilité des applications, ce point a de l'importance car l'application doit être consciente de son environnement changeant pour s'y adapter. Aussi la prise en compte des ressources est-elle une piste à explorer afin de mieux traiter la sûreté liée à l'auto-adaptation.

Evaluation du coût d'utilisation du service de sûreté

Une autre perspective à court terme concerne l'évaluation des performances liées à l'utilisation du service de sûreté. Le coût de l'utilisation du service de sûrement est semblable au coût d'utilisation de n'importe quel service. Cependant, le prototype du service n'a été testé qu'avec des

applications de petites tailles et peu d'adaptations. Il est donc nécessaire d'évaluer si le passage à l'échelle est possible aussi bien entre d'entités adaptées que du nombre d'adaptations réalisées. Ceci doit nous permettre de déterminer si le coût relatif à la vérification des préconditions d'une opération d'adaptation est raisonnable ou pas.

Amélioration de la précision dans la vérification des propriétés globales

La vérification des propriétés de sûreté se fait actuellement au moment où une adaptation est entamée. Cette forme de vérification est « statique » même si elle a lieu durant l'exécution d'une application dans le sens où elle n'a lieu que ponctuellement et n'a pas d'influence sur le flot d'exécution de l'application. Ainsi, les propriétés de sûreté ne sont vérifiées que sur les entités participant explicitement à une adaptation : le contexte d'utilisation de l'adaptation n'est alors pas pris en compte.

Par exemple, pour une fonctionnalité dont le comportement est modifié en faisant coopérer différentes entités, on ne connaît pas les paramètres d'appel effectifs et on ne connaît pas non plus la valeur de retour des fonctionnalités utilisées dans l'adaptation. Si certains paramètres ou certaines valeurs de retour sont utilisés dans l'adaptation (on appelle des fonctionnalités dessus), on contribue à créer un assemblage d'entités logicielles dont certaines ne sont pas connues au moment où l'adaptation est effectuée.

Cet aspect a un impact direct sur la précision des vérifications effectuées en ce qui concerne les propriétés de sûreté globales telle que la détection des cycles ou des points de non-déterminisme. En effet, certains cycles, par exemple, ne peuvent pas être détectés si lors de l'analyse du graphe de dépendance représentant l'assemblage créé par adaptation, celui-ci n'est pas complètement « instancié ». Dans ce cas, il peut être intéressant de contrôler les envois de messages afin de capturer le contexte d'exécution pour vérifier à nouveau ces propriétés globales. A ce niveau, il faut alors réfléchir à un compromis entre le coût des vérifications effectuées à l'envoi de messages et la précision des vérifications effectuées.

Aide à la prise de décision en terme d'adaptation

Le service de sûreté proposé fonctionne actuellement comme un oracle. C'est à dire qu'en fonction d'une adaptation donnée, il est capable de répondre si l'adaptation est sûre ou pas. La richesse du modèle Satin fait que l'on peut pousser plus loin l'utilisation du service. En effet, le service peut être étendu afin de pouvoir aider l'utilisateur à choisir au mieux comment adapter son application.

Cette proposition est particulièrement intéressante pour gérer les problèmes de déconnexions dans les applications mobiles : lors d'une déconnexion imprévue, le service pourrait proposer à l'utilisateur des actions d'adaptation à effectuer pour pallier aux problèmes de sûreté liés à la déconnexion.

Pilotage des adaptations depuis le service

Actuellement le service de sûreté n'est pas « intrusif » dans le sens où il n'a pas d'impact direct sur les applications qui l'utilisent : quelque soit le résultat de l'« évaluation » d'une adaptation par le service, la plate-forme sous-jacente peut décider d'effectuer l'adaptation de l'application car c'est toujours la plate-forme qui prend en charge l'adaptation.

Néanmoins, le service de sûreté valide la faisabilité d'une adaptation d'une application donnée en fonction des fonctionnalités offertes et des adaptations passées. Ceci implique que l'application soit partiellement représentée au niveau du service et donc que les adaptations soient également effectuées à ce niveau.

Ce point permet d'envisager une autre utilisation du service. Puisque les adaptations sont mises en œuvre dans le service pourquoi ne pas déléguer la prise en charge des adaptations au service. Cette proposition à l'avantage de faire bénéficier aux plates-formes existantes manquant d'expressivité en terme d'adaptabilité de nouveaux types d'adaptations.

Généralisation des approches proposées à d'autres modèles

Ce dernier point correspond à la perspective à plus long terme. Nous avons proposé un certain nombre de solutions pour la construction, la formalisation, la vérification et la transformation de modèles. Une voie à poursuivre dans ce domaine est d'évaluer dans quelles mesures ces approches sont réutilisables. Les point suivants sont plus des interrogations sur le potentiel de nos travaux pour l'évolution du domaine que des perspectives.

- L'approche puzzle peut elle être exploitée pour des modèles non exécutables et est elle un atout pour ce type de modèles ? Cette approche peut elle être utilisée pour la composition de modèles autrement que par collaboration ?
- La définition de propriétés de modèles sous forme de contraintes OCL est elle applicable pour tout modèle formel ? La traduction de ces propriétés en B est elle toujours possible ? Comment allier plus facilement les deux domaines ? Existe t-il d'autres formalismes plus appropriés à cette problématique tels que l'algèbre de processus ou la logique temporelle par exemple ?
- L'approche service peut elle être mise en œuvre quelque soit le modèle ?

Ces points pourraient étudiés en étendant le modèle Satin pour prendre en compte d'autres aspects de la sûreté de fonctionnement tels que la confidentialité ou l'intégrité par exemple.

Bibliographie

[1] J.-R. Abrial. *The B Book - Assigning Programs to Meanings*. Number ISBN 0-521-4961-5. Cambridge University Press, 1996.

[2] J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In *Proceedings of the 2nd Conference on the B method*, Montpellier, France, April 1998.

[3] J. Adamek and F. Plasil. Behavior protocols capturing errors and updates. In *Proceedings of USE*, University of Warsaw, Poland, 2003.

[4] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4 :32–54, 2005.

[5] J. Aldrich, C. Chambers, and D. Notkin. ArchJava : connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 187–197, New York, USA, Mai 19-25 2002. ACM Press.

[6] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, May 1997. CMU Technical Report CMU-CS-97-144.

[7] AOP Alliance. API AOP Alliance. http :/ /sourceforge.net/projects/aopalliance, 2005.

[8] J. P. A. Almeida. Dynamic reconfiguration of object-middleware-based distributed systems. Master's thesis, Dept. of Computer Science, University of Twente, Enschede, the Netherlands, June 2001.

[9] E. P. Anderson and T. Reenskaug. System design by composing structures of interacting objects. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 615 of *LNCS*, pages 133–153. Springer Verlag, 1992.

[10] O. Barais, L. Duchien, and L. Seinturier. Safarchie adl : Construire et déployer une architecture logicielle typée. Rapport technique 2004-10, Laboratoire d'Informatique Fondamentale de Lille, 2004.

[11] M. Beauvois. Brenda : Towards a composition framework for non-orhtogonal non-functional properties. In *Proceedings of DAIS 2003*, pages 29–40, 2003.

[12] L. Berger. *Mise en oeuvre des interactions en environnements distribués, compilés et fortement typés : le modèle MICADO*. Thèse de doctorat, Université de Nice-Sophia Antipolis, 2001.

[13] L. Bergmans and M. Aksit. Principles and design rationale of composition filters. In R. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, *Aspect-Oriented Software Development*. Addison-Wesley, 2004. ISBN 0-32-121976-.

[14] A. Beugnard. Une comparaison de langages objet relative au traitement de la redéfinition de méthode et à la liaison dynamique. In *Langages, Modèles, Objets (LMO'02)*, volume 8 of *L'objet*, Montpellier, France, 2002. Hermès Sciences.

[15] A. Beugnard, J.-M., N. Plouzeau, and D. Watkins. Making components contract aware. In *IEEE Software*, pages 38–45, 1999.

[16] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating, 2003.

[17] M. Blay-Fornarino, A. Charfi, D. Emsellem, A.-M. Pinna-Dery, and M. Riveill. Software interaction. *Journal of Object Technology*, 10(10), 2004.

[18] M. N. Bouraqadi-Saâdani. Un cadre réflexif pour la programmation par aspects. In *Proceedings of LMO'99*, Villefranche sur Mer, France, Janvier 1999. Hermès.

[19] M. N. Bouraqadi-Saâdani and T. Ledoux. Le point sur la programmation par aspects. *Technique et science informatiques*, 20(4) :489–512, 2001. Hermès.

[20] E. Bruneton, T. Coupaye, and J.-B. Stefani. Fractal Interface Specification 1.0, 2002.

[21] E. Bruneton, T. Coupaye, and J.-B. Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of WCOP*, 2002.

[22] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness : A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003 : Formal Methods : International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.

[23] J. Bézivin. On the unification power of models. *Software and System Modeling (SoSym)*, 4(2) :171–188, 2005.

[24] J. Bézivin, M. Didonet Del Fabro, F. Jouault, and P. Valduriez. Combining preoccupations with models. In *Proceedings of the First Workshop on Models and Aspects - Handling Crosscutting Concerns in Model-Driven Software Development (MDSD), ECOOP*, 2005.

[25] L. Cardelli. Type systems. *ACM Computing Surveys*, pages 263–264, 1996.

[26] C. Carrez, A. Fantechi, and E. Najm. Behavioural contracts for a sound composition of components. In Hartmut König, Monika Heiner, and Adam Wolisz, editors, *23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2003, IFIP TC 6/WG 6.1)*, volume 2767 of *LNCS*, pages 111–126. Springer-Verlag, Berlin, Germany, September 2003.

[27] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. 1re édition, avril 2000. chapitre 15, ISBN 2-84177-121-0.

[28] L. Chateigner, S. Chabridon, and G. Bernard. Intergiciel pour l'informatique nomade : réplication optimiste et réconciliation. In *Proceedings of MAJECSTIC*, Octobre 2003.

[29] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *ACM Transactions on Programming Languages and Systems*, volume 8, pages 244–263, April 1986.

[30] J. Clements, P. T. Graunke, S. Krishnamurthi, and M. Felleisen. Little languages and their programming environments. Monterey Workshop, 2001.

[31] P. Collet and R. Rousseau. Confract : un système pour contractualiser des composants hiérarchiques. Technical Report I3S/RR-2004-32-FR, Laboratoire I3S - Université de Nice-Sophia Antipolis, Mars 2004.

[32] K. Czarnecki and U. Eisenecker, editors. *Genererative Programming : Methods, Tools, and Applications*. Addison-Westley, 2000.

[33] M. Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution*, Tampa Bay, Florida, USA, 2001.

[34] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 141–150, March 2004. ACM Press.

[35] P. Dürr. Detecting semantic conflicts between aspects. Master's thesis, Dept. of Computer Science, University of Twente, Enschede, the Netherlands, April 2004.

[36] D. D'Souza and A. Wills. *Objects, Components and Frameworks With UML : The Catalysis Approach*. Addison-Westley, 1999.

[37] R. Ducournau. Spécialisation et sous-typage : thème et variations. *Technique et science informatiques*, 2002.

[38] J. K. Filipe. A logic-based formalisation for component specification. *Journal of Object Technology, Special issue : TOOLS USA 2002 proceedings.*, 1(3) :231–248, 2002.

[39] R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. In *Proceedings of FSE*, 2001.

[40] A. Flissi and P. Merle. Une démarche dirigée par les modèles pour construire les machines de déploiement des intergiciels à composants. *Conférence Langages et Modèles à Objets (LM0 2005)*, 11, 2005.

[41] A. Flissi, P. Merle, and C. Gransart. A component-based software infrastructure for ubiquitous computing. In *The 4th International Symposium on Parallel and Distributed Computing (ISPDC 2005)*, Lille, France, 4-6 July 2005. (To Appear).

[42] C. Francisco N. Garcia. Compose * : A runtime for the .Net platform. Master's thesis, Dept. of Computer Science, University of Twente, Enschede, the Netherlands, August 2003.

[43] D. Garlan, R. Monroe, and D. Wile. ACME : An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, USA, Novembre 1997.

[44] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Journal on Software and System Modeling*, 2005. To Appear.

[45] J. Gorinsek, S. Van Baelen, Y. Berbers, and K. De Vlaminck. Managing quality of service during evolution using component contracts. In G. Kniesel, P. Costanza, and J. L. Fiadeiro, editors, *ETAPS 2003 Workshop on Unanticipated Software Evolution (USE2003)*, Warsaw, Poland, 2003.

[46] M. M. Gorlick and R. R. Razouk. Using weaves for software construction and analysis. In *Proceedings of the 13th International Conference on Software Engineering (ICSE13)*, pages 23–34, Austin, TX, USA, May 1991.

[47] J. Greenfield and K. Short, editors. *Software Factories : Assembling Applications with Patterns, Frameworks, Models and Tools*. John Wiley and Sons, 2004.

[48] O. Hachani and D. Bardou. Un rapprochement d'aspectj et d'hyper/j par les méta-modèles. In *Actes de la première Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2004)*, IRCAM Paris, France, 2004.

[49] J. Hanneman, R. Chitchyan, and A. Rashid. Analysis of aspect-oriented software (aaos) workshop report. Technical report, University of California, Darmstadt, Germany, 2003.

[50] G. Heineman and W. Councill, editors. *Component-Based Software Engineering, Putting the Pieces Together*. Addison-Westley, 2001. *ISBN : 0-201-70485-4*.

[51] D. Hoareau and Y. Mahéo. Distribution d'un composant hiérarchique dans un environnement partiellement connecté. In *Actes des Journées Composants*, 2005.

[52] IS0. Open Distributed Processing Reference Model : Architectural semantics. ITU-T | ISO/IEC Recommendation X.904 | International Standard 10746-4, 1995.

[53] IS0. Open Distributed Processing Reference Model : Architecture. ITU-T | ISO/IEC Recommendation X.903 | International Standard 10746-3, 1995.

[54] IS0. Open Distributed Processing Reference Model : Foundations. ITU-T | ISO/IEC Recommendation X.902 | International Standard 10746-2, 1995.

[55] IS0. Open Distributed Processing Reference Model : Overview. ITU-T | ISO/IEC Recommendation X.901 | International Standard 10746-1, 1995.

[56] J.-C. Laprie et Al. *Guide de la Sûreté de Fonctionnement*. 1996.

[57] Jboss. Jboss home page. http ://www.jboss.org/, 2005.

[58] J. Jing, A. S. Helal, and A. Elmagarmid. Client-server computing in mobile environments. *ACM Computing Surveys*, 31(2) :117–157, 1999.

[59] G. Kiczales and J. Lamping. Aspectj home page. http ://eclipse.org/aspectj, 2001.

[60] G. Kniesel and T. Rho. Generic aspect languages - needs, options and challenges. *Actes JFDLPA'05*, 2005.

[61] P. Koopmans. On the design and implementation of the sina/st language. Master's thesis, Dept. of Computer Science, University of Twente, August 1995.

[62] N. Kouici, D. Conan, and G. Bernard. Intégration d'un service de gestion de déconnexions dans les conteneurs des composants. In *Actes des Journées Composants*, 2004.

[63] N. Kouici, N. Sabri, D. Conan, and G. Bernard. Mada, une approche pour le développement d'applications mobiles. In *Proceedings of ACM UbiMob*, Nice, France, 1-3 juin 2004.

[64] H. Kreger. Web services conceptual architecture (wsca 1.0). IBM Software Group, 2001.

[65] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML : A behavioral interface specication language for java. Technical Report 98-06i, Iowa State University, Department of Computer Science, February 2000.

[66] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM Symp. Principles of Programming Languages (POPL'85)*, pages 97–107, New Orleans, LA, USA, 1985.

[67] B. Liskov and J. Wing. A behavioral notion of sub-typing. *ACM Transactions on Programming Languages and Systems*, 16(6) :1811–1841, 1994.

[68] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, pages 717–734, September 1995.

[69] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of ACM SIGSOFT'96 : Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pages 3–14, San Francisco, CA, USA, October 1996.

[70] V. Marangozova and D. Hagimont. Adaptation d'une application répartie pour la disponibilité : Expérience et évaluation. In *RENPAR/CFSE/SYMPA*, 2001.

[71] R. Marvie. Picore metamodeling environment web page. http ://www.lifl.fr/marvie/Software.html#picore, 2004.

[72] R. Marvie, P. Merle, J.-M. Geib, and M. Vadet. OpenCCM : une plate-forme pour composants CORBA. In *Actes de la seconde conférence française sur les systèmes d'exploitation (CFSE-2)*, Paris, France, 2001.

[73] N. McEachen and R. T. Alexander. Distributing classes with woven concerns : an exploration of potential fault scenarios. In *AOSD '05 : Proceedings of the 4th international conference on Aspect-oriented software development*, pages 192–200, New York, NY, USA, 2005. ACM Press.

[74] N. Medvidovic. Adls and dynamic architecture changes. In ed. A. L. Wolf, editor, *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 24–27, San Francisco, USA, October 1996.

[75] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *Proceedings of ACM SIGSOFT'96 : Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pages 24–32, San Francisco, CA, USA, October 1996.

[76] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1) :70–93, 1997.

[77] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22th International Conference on Software Engineering*, Limerick, Ireland, 2000.

[78] B. Meyer. Applying "design by contract". In *IEEE Computer*, volume 25, pages 40–51. 1992.

[79] Microsoft. Microsoft .NET platform. http :/ / www.microsoft.com / net / , February 2001.

[80] P. Mulet, J. Malenfant, and P. Cointe. Towards a methodology for explicit composition of metaobjects. In *Proceedings of OOPSLA'95*, pages 316–330, Austin, Texas, 1995.

[81] A. Muller, O. Caron, B. Carre, and G. Vanwormhoudt. On some properties of parameterized model application. In A. Hartman and editors D. Kreische, editors, *ECMDA-FA 2005*, volume 3748 of *LNCS*, pages 130–144. Springer-Verlag, 2005.

[82] E. Najm and A. Nimour. Explicit behavioral typing for object interfaces. In *Proceedings of SOAP'99 workshop*, Lisbon, Portugal, June 99. BRICS.

[83] O. Nano and M. Blay-Fornarino. Annotations et transformations de modèles pour l'intégration de services. *Conférence Langages et Modèles à Objets - LM0 2004, Lille France - publié dans la revue RTSI - série l'Objet (Lavoisier Eds) - ISBN : 2-7462-0887-3*, 10(2–3) :175–188, 2004.

[84] Y. V. Natis. Service-oriented architecture scenario. Gartner, Inc, 2003.

[85] O. Nierstrasz. Contractual types. Technical report, Institut für Informatik und Angewandte Mathematik, University of Bern, August 2003.

[86] Objectweb Consortium. The Fractal Component Model. http :/ / fractal.objectweb.org / .

[87] Objectweb Consortium. JOnAS : Java (TM) Open Application Server. http :/ / jonas.objectweb.org / , 2005.

[88] OMG. Meta object facility specification. OMG Document AD / 97-08-14, 1997.

[89] OMG. CORBA 3.0 New Components Chapters. OMG Document ptc / 2001-11-03, 2001.

[90] OMG. Model Driven Architecture. OMG Document ormsc / 2001-07-01, 2001.

[91] OMG. Unified Modeling Language Specification. OMG Document formal / 03-03-01, 2003.

[92] OMG. Common Object Request Broker Architecture : Core specification. OMG TC Document formal / 04-03-01, 2004.

[93] Open Services Gateway initiative. OSGi service platform (3d release). http :/ / www.osgi.org / , 2003.

[94] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 177–186, Kyoto, Japan, April 1998.

[95] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theor. Pract. Object Syst.*, 2(3) :179–202, 1996.

[96] R. Pawlak, L. Duchien, and L. Seinturier. Compar : Ensuring safe around advice composition. In *Proceedings of the 7th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS 2005*, Athens, Greece, June 2005.

[97] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC : A flexible and efficient solution for aspect-oriented programming in java. In A. Yonezawa and S. Matsuoka, editors, *Reflection*, volume 2192 of *LNCS*, pages 1–24. Springer-Verlag, 2001.

[98] F. Peschanski. A versatile event-based communication model for generic distributed interactions. In *ICDCS Workshops*, pages 503–510, 2002.

[99] N. Pessemier. Fractal Aspect Component. http :/ / www.lifl.fr / pessemie / FAC / , 2005.

[100] N. Pessemier, L. Seinturier, and L. Duchien. Components, ADL and AOP : Towards a common approach. In *In Workshop ECOOP Reflection, AOP and Meta-Data for Software Evolution (RAM-SE04)*, 2004.

[101] F. Plasil, D. Balek, and R. Janecek. SOFA / DCUP : Architecture for component trading and dynamic updating. In *Proceedings of ICCDS'98*, Annapolis, Maryland, USA, 1998.

[102] M. Richters. The USE tool : A UML-based specification environment. http : / / www.db.informatik.uni-bremen.de / projects / USE / .

[103] M. Rits, K. Boudaoud, and M. Riveill. Security engineering for adaptable software components. In *First ACM Workshop on Business Driven Security Engineering (BIZSEC)*, Faifax, USA, 31 october 2003.

[104] E. Roman, S. W. Ambler, and T. Jewell. *Mastering Enterprise Java Beans II and the Java 2 Platform*. John-Wiley & Sons Inc., enterprise edition, 2002.

[105] J. Rothenberg. The nature of modeling. *Artificial Intelligence, Simulation, and Modeling*, pages 75–92, 1989.

[106] H. Roussain and F. Guidec. Déploiement de composants logiciels sur des équipements mobiles communicants : une approche coopérative. In *Actes des Journées Composants*, 2005.

[107] P. Salinas. Adding systemic crosscutting and super-imposition to composition filters. Master's thesis, Dept. of Computer Science, University of Twente, Vrije Universiteit Brussel, August 2001.

[108] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Symposium on Principles of Distributed Computing*, pages 1–7, 1996.

[109] M.-T. Segarra and F. André. A framework for dynamic adaptation in wireless environments. In *Proceedings of TOOLS Europe 2000*, Mont St. Michel, St. Malo, France, 2000.

[110] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, pages 314–335, 1995.

[111] Softeam. The objecteering / UML tool suite. http : / / www.objecteering.com / , 2002.

[112] G. Straw, G. Georg, E. Song, S. Ghosh, R. France, and J. M. Bieman. Model composition directives. In T. Baar, A. Strohmeier, A. Moreira, and editors S. J. Mellor, editors, *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference*, volume 3273 of *LNCS*, pages 84–97, Lisbon, Portugal, 2004. Springer.

[113] C. Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Westley, 1999. *ISBN : 0-201-17888-5*.

[114] F. Taïani, M.-O. Killijian, and J.-C. Fabre. Intergiciels pour la tolérance aux fautes : état de l'art et défis. *Technique et science informatiques*, 2005. Éditions Hermès, accepté pour publication.

[115] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual*. IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, 2000.

[116] L. Temal and D. Conan. Détections de défaillances, de connectivité et de déconnexion. In *Proceedings of ACM UbiMob*, Nice, France, 1-3 juin 2004.

[117] F. Tessier, L. Badri, and M. Badri. Towards a formal detection of semantic conflicts between aspects : A model-based approach. In *Proceedings of the 5 th Aspect-Oriented Modeling Workshop in conjunction with UML 2004*, Lisbon, Portugal, 2004.

[118] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages : An annotated bibliography. *SIGPLAN Notices*, 35(6) :26–36, 2000.

[119] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st IEEE Symp. Logic in Computer Science (LICS'86)*, pages 332–344, Cambridge, MA, USA, June 1986.

[120] J. Warmer and A. Kleppe. OCL : The constraint language of the UML. *Journal of Object-Oriented Programming*, 1999.

[121] P. Wegner and S. R. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In S. Gjessing and K. Nygaard, editors, *Proceedings of ECOOP'88*, volume 322 of *LNCS*, pages 55–77. Springer-Verlag, 1988.

[122] J.C. Wichman. ComposeJ : The development of a preprocessor to facilitate composition filters in the Java language. Master's thesis, Dept. of Computer Science, University of Twente, Enschede, the Netherlands, December 1999.

[123] R. Wiebicke. Utility support for checking ocl business rules in java programs. Master's thesis, TU-Dresden, December 2000.

[124] G. Zelesnik. The unicon language reference manual. Technical report, School of Computer Science, Carnegie Mellon Universtity, Pittsburgh, USA, May 1996.

Quatrième partie

Annexes

A
Grammaire de ASL

La syntaxe concrète du langage ASL permet de définir un schéma d'adaptations. Nous utilisons une notation BNF avec les conventions suivantes. Les symboles terminaux sont des chaînes de caractères entourées de doubles guillemets. Toutes les autres chaînes de caractères représentes des symboles non terminaux. Les éléments d'une séquence sont séparés par des blancs, les alternatives sont séparées par le symbole "|". Un élément optionnel est entouré par "[" et "]", un élément répété zéro ou plusieurs fois est entouré par "{" et "}".

```
root := "adaptationPattern" ident "(" formalParameters")" "{" adaptations "}"
adaptations := adaptation { "," adaptation }

adaptation := "newRole" typeName "on" ident "->" methodDeclarations
            | "newPort" [ "!" ] methodDeclaration "->" exp
            | "modifyPort" [ "!" ] methodDeclaration "->" exp

methodDeclarations:= methodDeclaration { ";" methodDeclaration }
methodDeclaration := ident "." signature
signature := ident "(" [ formalParameters ] ")" [ ":" typeName ]
call := ident "(" [ effectiveParameters | const ] ")"

exp := expBloc
     | expBloc ";" exp
     | expBloc "//" exp

expBloc := "if" cond "then" exp "else" exp "endif"
         | [ decl "=" ] ident "." call
         | "result" ident "." call
         | "throw" typeName
         | "(" exp ")"

cond := ident "." call
formalParameters:= decl { "," decl }
```

```
effectiveParameters := param { "," param }
decl := typeName ident
param := ident | const
typeName := "A"-"Z" { "a"-"z" | "0"-"9" | "A"-"Z" | "_" }
methodName := ( "*" | ident ) { ident } [ "*" ]
ident := "a"-"z" { "a"-"z" | "0"-"9" | "A"-"Z" | "_" }
```

B

Contraintes OCL du modèle Satin

```
context Template::createFromFacets(roles : Collection(AbstractRole),
                                   implantation : Implantation) : Template
pre C2a :
  roles->forAll(r1, r2 | not (r1 = r2) implies
  r1.providedPorts->intersection(r2.providedPorts) = oclEmpty(Set(Port)))


context Template::createFromViewPoints(roles : Sequence(AbstractRole),
                                       implantations : Sequence(Implantation))
                                      : Template
pre C2b :
  roles->forAll(r1, r2 | not (r1 = r2) implies
  r1.getProvidedPorts()->intersection(r2.getProvidedPorts()) = oclEmpty(Set(Port)))


context Component::replace(cNew : Component)
pre C3:
   self.roles->forAll(r1 | cNew.roles->exists(r2 | r2.isSubRoleOf(r1)))


context AdaptationPattern::createFrom(roles : Sequence(GenericRole),
                                      adaptations : Sequence(ElementaryAdaptation))
                                     : AdaptationPattern
pre C5 :
  adaptations->forAll(a1, a2 | (a1.pointcut().filters(a2.pointcut()) or
  a2.pointcut().filters(a1.pointcut())) implies a1.isCompatibleWith(a2))

pre C6 :
  adaptations->forAll(a | not self.containsNDP(a, roles, adaptations))

pre C7 :
  adaptations->forAll(a |
  not self.containsCycleFrom(a, oclEmpty(Set(Port)), roles, adaptations))
```

```
context AdaptationPattern::instantiate(components : Sequence(Component))
                                      : AdaptationInstance
pre C9 :
  self.elementaryadaptations->forAll(a |
  components->at(a.position()).getAdaptationPorts()->forAll(ap |
  a.pointcut().filters(ap.portToAdapt) implies a.isCompatibleWith(ap.adaptation) ))

pre C10 :
  self.elementaryadaptations->forAll(a | not self.containsNDP2(a))

pre C11 :
  self.elementaryadaptations->forAll(a | self.containsCycleFrom2(a, oclEmpty(Set(Port))))

pre C14 :
  Sequence{1..components->size}->forAll(index : Integer |
  components->at(index).roles->exists(r | self.parameters->at(index).filters(r)))


context AdaptationInstance::remove()
pre C16:
  self.assignedadaptations->forAll(a | not self.containsNDP(a))

pre C17:
  self.assignedadaptations->forAll(a |
  not self.containsCycleFrom(a.getProperties(), oclEmpty(Set(Port))))
```

C

Implémentation de testCreateTemplate

```
procedure testCreateTemplate(portnamelist: Set(String), nbr: Integer,
                             nbp : Integer, nbf : Integer)
  var t: Template, lesroles: Sequence(AbstractRole), impl: Implantation,
  pplist: Sequence(ProvidedPort), flist : Sequence(ProvidedPort),
  exp: Expression, tmp: Set(String), tmp2 : Sequence(String), i : Integer, irole : AbstractRole;

begin
  t := Create(Template);
  impl := Create(Implantation);
  lesroles := CreateN(AbstractRole, [nbr]);
  flist := CreateN(ProvidedPort, [nbf]);
  irole := Create(AbstractRole);
  Insert(implantation_role, [impl], [irole]);

  i := [1];
  for r:AbstractRole in [lesroles] begin
    [r].roleName := [Sequence{'r1', 'r2', 'r3', 'r4', 'r5'}->at(i)];
    pplist := CreateN(ProvidedPort, [nbp]);
    tmp := [oclEmpty(Set(String))];
    for pp:ProvidedPort in [pplist] begin
      exp := Create(Expression);
      tmp2 := Sub([(portnamelist-tmp)->asSequence], [(portnamelist-tmp)->size]);
      [exp].s := Any([tmp2]);
      Insert(baseport_expression, [pp], [exp]);
      Insert(role_providedport, [r], [pp]);
      tmp := [tmp->including(exp.s)];
    end;
    Insert(template_role, [t], [r]);
    i := [i+1];
  end;
```

```
  tmp := [oclEmpty(Set(String))];
  for pp:ProvidedPort in [flist] begin
    exp := Create(Expression);
    tmp2 := Sub([(portnamelist-tmp)->asSequence], [(portnamelist-tmp)->size]);
    [exp].s := Try([tmp2]);
    Insert(baseport_expression, [pp], [exp]);
    Insert(role_providedport, [irole], [pp]);
    tmp := [tmp->including(exp.s)];
  end;

  Insert(template_implantation, [t], [impl]);
end;
```

D

Machine B pour une partie du modèle Satin

```
MACHINE
    satin

SETS
  TEMPLATES; IMPLANTATIONS; COMPONENTS;
  ADAPTATIONPATTERNS; ADAPTATIONINSTANCES; ELEMENTARYADAPTATIONS;
  GENERICROLES; ABSTRACTROLES; CONCRETEROLES;
  PROVIDEDPORTS; EMITTEDPORTS; EMITTEDPORTS2

ABSTRACT_CONSTANTS
  ISSUBROLEOF, EQUALS, FILTERS

PROPERTIES
  ISSUBROLEOF : ABSTRACTROLES <-> ABSTRACTROLES &
  EQUALS : PROVIDEDPORTS <-> PROVIDEDPORTS &
  id(PROVIDEDPORTS) <: EQUALS &
  EQUALS = EQUALS~ &
  FILTERS : PROVIDEDPORTS <-> PROVIDEDPORTS

VARIABLES
  Templates, Components, AdaptationPatterns, AdaptationInstances,
  HasTemplate, HasRoles, HasImplantations, HasImplRole,
  HasPattern, HasGenericRoles, HasElementaryAdaptations,
  ApplyPosition, HasProperties, HasPointcut, HasType,
  HasParticipants,
  HasFunctionalities, HasConcreteRoles,
  HasProvidedPorts, HasEmittedPorts,
  HasTarget, HasPortToEmit,
  HasAdaptationPorts,
  HasProvidedPorts2,
  HasProvidedPorts3, HasEmittedPorts3,
  HasTarget2, HasPortToEmit2,
  HasAddedEmittedPorts
```

```
DEFINITIONS
  HasInstances == HasPattern~;
  HasAdaptations == HasParticipants~

INVARIANT
  Templates <: TEMPLATES &

  Components <: COMPONENTS &
  AdaptationPatterns <: ADAPTATIONPATTERNS &
  AdaptationInstances <: ADAPTATIONINSTANCES &

  HasTemplate : COMPONENTS +-> TEMPLATES &
  dom(HasTemplate) = Components &
  HasRoles : TEMPLATES <-> ABSTRACTROLES &
  dom(HasRoles) = Templates &

  HasImplantations : TEMPLATES <-> IMPLANTATIONS &
  dom(HasImplantations) = Templates &
  HasImplRole : IMPLANTATIONS +-> ABSTRACTROLES &

  HasPattern : ADAPTATIONINSTANCES +-> ADAPTATIONPATTERNS &
  dom(HasPattern) = AdaptationInstances &

  HasGenericRoles : ADAPTATIONPATTERNS +-> seq(GENERICROLES) &
  dom(HasGenericRoles) = AdaptationPatterns &

  HasElementaryAdaptations : ADAPTATIONPATTERNS <-> ELEMENTARYADAPTATIONS &
  dom(HasElementaryAdaptations) = AdaptationPatterns &

  ApplyPosition : ELEMENTARYADAPTATIONS +-> INTEGER &

  HasProperties : ELEMENTARYADAPTATIONS <-> EMITTEDPORTS2 &
  /* On se limite à controler les ports fournis */
  HasPointcut : ELEMENTARYADAPTATIONS +-> PROVIDEDPORTS &
  /* Les chaines de caracteres ne sont pas autorisées ici
     On utilise les entiers à la place :
     "add" => 0, "control" => 1 */
  HasType : ELEMENTARYADAPTATIONS +-> INTEGER &

  HasParticipants : ADAPTATIONINSTANCES +-> seq(COMPONENTS) &
  dom(HasParticipants) = AdaptationInstances &
  !adi.(adi:AdaptationInstances =>
  ran(HasParticipants(adi))<:Components) &

  HasFunctionalities : IMPLANTATIONS <-> PROVIDEDPORTS &

  HasConcreteRoles : COMPONENTS <-> CONCRETEROLES &
  dom(HasConcreteRoles) = Components &
  HasConcreteRoles~ : CONCRETEROLES +-> COMPONENTS &

  HasProvidedPorts : CONCRETEROLES <-> PROVIDEDPORTS &
  HasEmittedPorts : CONCRETEROLES <-> EMITTEDPORTS &
  HasTarget : EMITTEDPORTS +-> COMPONENTS &
  HasPortToEmit : EMITTEDPORTS +-> PROVIDEDPORTS &

  HasAdaptationPorts : CONCRETEROLES <-> PROVIDEDPORTS &
```

```
  HasProvidedPorts2 : ABSTRACTROLES <-> PROVIDEDPORTS &

  HasProvidedPorts3 : GENERICROLES <-> PROVIDEDPORTS &
  HasEmittedPorts3 : GENERICROLES <-> EMITTEDPORTS2 &
  HasTarget2 : EMITTEDPORTS2 +-> INTEGER &
  HasPortToEmit2 : EMITTEDPORTS2 +-> PROVIDEDPORTS &

  HasAddedEmittedPorts : ADAPTATIONINSTANCES <-> EMITTEDPORTS &

  /* Propriété P1a */
  !(t,r).(t: Templates & r:ABSTRACTROLES & r:HasRoles[{t}] =>
  HasImplRole[HasImplantations[{t}]]*{r} <: ISSUBROLEOF) &
  !(c,pp1).(c:Components & pp1:PROVIDEDPORTS & pp1:HasProvidedPorts[HasConcreteRoles[{c}]] =>
  #pp2.(pp2:PROVIDEDPORTS &
        pp2:HasProvidedPorts2[HasRoles[HasTemplate[{c}]]] & pp1|->pp2 : EQUALS)
  or
  #pp3.(pp3:PROVIDEDPORTS &
        pp3:HasAdaptationPorts[HasConcreteRoles[{c}]] & pp1|->pp3 : EQUALS)) &

  /* Propriété P3 */
  !ep.(ep:EMITTEDPORTS & ep:HasEmittedPorts[HasConcreteRoles[Components]]  =>
  #pp.(pp:PROVIDEDPORTS & pp:HasProvidedPorts[HasConcreteRoles[HasTarget[{ep}]]] &
       HasPortToEmit[{ep}]*{pp} <: FILTERS))

INITIALISATION
  Templates := {} ||
  Components := {} ||
  AdaptationPatterns := {} ||
  AdaptationInstances := {} ||

  HasTemplate := {} ||
  HasRoles := {} ||
  HasImplantations := {} ||
  HasImplRole := {} ||

  HasPattern := {} ||
  HasGenericRoles := {} ||
  HasElementaryAdaptations  := {} ||
  ApplyPosition := {} ||
  HasProperties := {} ||
  HasPointcut := {} ||
  HasType := {} ||

  HasParticipants := {} ||

  HasFunctionalities := {} ||
  HasConcreteRoles := {} ||

  HasProvidedPorts := {} ||
  HasEmittedPorts := {} ||
  HasTarget := {} ||
  HasPortToEmit := {} ||
  HasAdaptationPorts := {} ||

  HasProvidedPorts2 := {} ||
```

```
  HasProvidedPorts3 := {} ||
  HasEmittedPorts3 := {} ||
  HasTarget2 := {} ||
  HasPortToEmit2 := {} ||

  HasAddedEmittedPorts := {}

OPERATIONS
  /* On se limite à créer des templates à partir de rôles abstraits et d'une implantation */
  createTemplateFromFacets(roles,impl) =
    PRE
      /* Contraintes de typage et de cardinalité */
      roles <: ABSTRACTROLES &
      impl : IMPLANTATIONS &
      roles /= {} &
      /* Les rôles et l'implantation sont correctement définis */
      !r.(r : roles => HasProvidedPorts2[{r}] /= {}) &
      HasFunctionalities[{impl}] /= {} &
      /* On peut encore créer des templates */
      TEMPLATES - Templates /= {} &
      /* Hypothèse H1 préservant P1a : correspond à la contrainte OCL C1a */
      !r.(r:ABSTRACTROLES & r:roles => HasImplRole[{impl}]*{r} <: ISSUBROLEOF)
      THEN
        ANY t WHERE t:TEMPLATES-Templates
        THEN
            Templates := Templates\/{t} ||
            HasRoles := HasRoles \/ {t}*roles ||
            HasImplantations := HasImplantations \/ {t|->impl}
      END
    END;

  InstantiateTemplate(template, crl) =
    PRE
      /* Contraintes de typage et de cardinalité */
      template : Templates &
      crl <: CONCRETEROLES &
      crl /= {} &
      /* Les rôles contenus dans crl ne sont associés à aucun composant existant */
      crl /\ ran(HasConcreteRoles) = {} &
      /* Les rôles contenus dans crl sont correctement définis */
      crl <: dom(HasProvidedPorts) &
      /* On peut encore créer des composants */
      COMPONENTS - Components /= {} &
      /* Hypothèse H2 préservant P1a */
      !pp1.(pp1:PROVIDEDPORTS & pp1:HasProvidedPorts[crl] =>
      #pp2.(pp2:PROVIDEDPORTS & pp2:HasProvidedPorts2[HasRoles[{template}]]
            & pp1|->pp2:EQUALS)) &
      /* Hypothèse H3 préservant P3 */
      HasEmittedPorts[crl] = {}
    THEN
      ANY c WHERE c:COMPONENTS-Components
      THEN
          Components := Components \/ {c} ||
          HasTemplate := HasTemplate \/ {c|->template} ||
          HasConcreteRoles := HasConcreteRoles \/ {c}*crl
```

```
    END
  END;

createPattern(roles, adaptations) =
  PRE
    /* Contraintes de typage et de cardinalité */
    roles : seq(GENERICROLES) &
    card(roles) /= 0 &
    adaptations <: ELEMENTARYADAPTATIONS &
    card(adaptations) /= 0 &
    /* On peut encore créer des schémas */
    ADAPTATIONPATTERNS-AdaptationPatterns /= {} &
    /* Les adaptations élémentaires sont correctement définies */
    adaptations <: dom(HasProperties) &
    adaptations <: dom(HasPointcut) &
    HasProperties[adaptations] <: dom(HasTarget2) &
    HasProperties[adaptations] <: dom(HasPortToEmit2) &
    /* Les rôles sont bien déduits des adaptations élémentaires */
    !(ea,ep).(ea:ELEMENTARYADAPTATIONS & ep : EMITTEDPORTS2 &
              ea:adaptations & ep : HasProperties[{ea}] =>
         HasTarget2(ep) : dom(roles) &
         HasPortToEmit2[{ep}] <: HasProvidedPorts3[{roles(HasTarget2(ep))}] &
         ep : HasEmittedPorts3[{roles(ApplyPosition(ea))}])
  THEN
    ANY adp WHERE adp:ADAPTATIONPATTERNS-AdaptationPatterns
    THEN
        AdaptationPatterns := AdaptationPatterns\/{adp} ||
        HasGenericRoles := HasGenericRoles\/{adp|->roles} ||
        HasElementaryAdaptations := HasElementaryAdaptations\/{adp}*adaptations
    END
  END;

instantiatePattern(pattern, components, epset) =
  PRE
    /* Contraintes de typage et de cardinalité */
    pattern : AdaptationPatterns &
    components : seq(Components) &
    card(components) /= 0 &
    epset <: EMITTEDPORTS &
    card(epset) /= 0 &
    /* Les ports émis contenus dans epset sont correctement définis */
    HasPortToEmit2[HasProperties[HasElementaryAdaptations[{pattern}]]] <: HasPortToEmit[epset] &
    HasTarget[epset] <: ran(components) &
    !ep1.(ep1:EMITTEDPORTS2 & ep1:HasEmittedPorts3[ran(HasGenericRoles(pattern))] =>
    #ep2.(ep2:EMITTEDPORTS & ep2:epset
                           & HasPortToEmit[{ep2}] = HasPortToEmit2[{ep1}]
                           & components(HasTarget2(ep1)) : HasTarget[{ep2}])) &
    /* On applique le schéma au bon nombre de composants */
    dom(HasGenericRoles(pattern))=dom(components) &
    /* On peut encore créer des instances de schémas d'adaptation */
    ADAPTATIONINSTANCES-AdaptationInstances /= {} &
    /* Hypothèse H4 préservant P3 : correspond à la contrainte OCL C8 */
    !ep.(ep:EMITTEDPORTS & ep:epset =>
    #pp.(pp:PROVIDEDPORTS & pp:HasProvidedPorts[HasConcreteRoles[HasTarget[{ep}]]]
                          & HasPortToEmit[{ep}]*{pp} <: FILTERS))
  THEN
```

```
    ANY adi
    WHERE adi:ADAPTATIONINSTANCES-AdaptationInstances
    THEN
        AdaptationInstances := AdaptationInstances\/{adi} ||
        HasPattern := HasPattern\/{adi|->pattern} ||
        HasParticipants := HasParticipants\/{adi|->components} ||
        HasAddedEmittedPorts := HasAddedEmittedPorts\/{adi}*epset ||

        HasProvidedPorts := HasProvidedPorts\/
           UNION ea.(ea:ELEMENTARYADAPTATIONS & ea:HasElementaryAdaptations[{pattern}] &
                     HasType[{ea}] = {0} |
           HasConcreteRoles[{components(ApplyPosition(ea))}]*HasPointcut[{ea}]) ||

        HasEmittedPorts := HasEmittedPorts\/
           UNION(ea,ep1,ep2).
                (ea:ELEMENTARYADAPTATIONS & ep1:EMITTEDPORTS2 & ep2:EMITTEDPORTS &
                 ea:HasElementaryAdaptations[{pattern}] &
                 ep1:HasEmittedPorts3[{HasGenericRoles(pattern)(ApplyPosition(ea))}] &
                 ep2:epset & HasPortToEmit2[{ep1}]*HasPortToEmit[{ep2}] <: FILTERS &
                 components(HasTarget2(ep1)):HasTarget[{ep2}] |
            HasConcreteRoles[{components(ApplyPosition(ea))}]*{ep2}) ||

        HasAdaptationPorts := HasAdaptationPorts\/
           UNION ea.(ea:ELEMENTARYADAPTATIONS & ea:HasElementaryAdaptations[{pattern}] |
           HasConcreteRoles[{components(ApplyPosition(ea))}]*HasPointcut[{ea}])
    END
  END;

removeInstance(adaptationinst) =
  PRE
    /* Contraintes de typage */
    adaptationinst : AdaptationInstances &
    /* Hypothèse H5 préservant P1a */
    !(c, pp1).(c:Components & pp1:PROVIDEDPORTS &
          pp1:(HasProvidedPorts -
          UNION ea.(ea:ELEMENTARYADAPTATIONS &
                    ea:HasElementaryAdaptations[HasPattern[{adaptationinst}]] & HasType[{ea}] = {0} |
          HasConcreteRoles[{HasParticipants(adaptationinst)(ApplyPosition(ea))}]*
          HasPointcut[{ea}]))[HasConcreteRoles[{c}]] =>
    #pp2.(pp2:PROVIDEDPORTS &
          pp2:(HasProvidedPorts2[HasRoles[HasTemplate[{c}]]]\/
               (HasAdaptationPorts -
                UNION ea.(ea:ELEMENTARYADAPTATIONS &
                          ea:HasElementaryAdaptations[HasPattern[{adaptationinst}]] |
                HasConcreteRoles[{HasParticipants(adaptationinst)(ApplyPosition(ea))}]*
                HasPointcut[{ea}]))[HasConcreteRoles[{c}]]) &
    pp1 |-> pp2 : EQUALS)) &
    /* Hypothèse H6 préservant P3 : correspond à la contrainte OCL C14 */
    !(ea,ep).(ea:ELEMENTARYADAPTATIONS & ep:EMITTEDPORTS &
              ea:HasElementaryAdaptations[HasPattern[{adaptationinst}]] &
              HasType[{ea}] = {0} &
              ep:HasEmittedPorts[HasConcreteRoles[Components]] &
              HasParticipants(adaptationinst)(ApplyPosition(ea)):HasTarget[{ep}] =>
              HasPointcut[{ea}]/\HasPortToEmit[{ep}] = {}) &
    !ep.(ep:EMITTEDPORTS & ep:HasEmittedPorts[HasConcreteRoles[Components]] =>
    #pp.(pp:PROVIDEDPORTS &
```

```
          pp:(HasProvidedPorts -
              UNION ea.(ea:ELEMENTARYADAPTATIONS &
                        ea:HasElementaryAdaptations[HasPattern[{adaptationinst}]] &
                        HasType[{ea}] = {0} |
              HasConcreteRoles[{HasParticipants(adaptationinst)(ApplyPosition(ea))}]*
              HasPointcut[{ea}]))[HasConcreteRoles[HasTarget[{ep}]]] &
      HasPortToEmit[{ep}]*{pp} <: FILTERS))
    THEN
        AdaptationInstances := AdaptationInstances - {adaptationinst} ||
        HasPattern := {adaptationinst} <<| HasPattern ||
        HasParticipants := {adaptationinst} <<| HasParticipants ||
        HasAddedEmittedPorts := {adaptationinst} <<| HasAddedEmittedPorts ||

        HasProvidedPorts := HasProvidedPorts -
             UNION ea.(ea:ELEMENTARYADAPTATIONS &
                       ea:HasElementaryAdaptations[HasPattern[{adaptationinst}]] &
                       HasType[{ea}] = {0} |
             HasConcreteRoles[{HasParticipants(adaptationinst)(ApplyPosition(ea))}]*HasPointcut[{ea}]) ||

        HasEmittedPorts := HasEmittedPorts -
             UNION(ea,ep1,ep2).
                  (ea:ELEMENTARYADAPTATIONS & ep1:EMITTEDPORTS2 & ep2:EMITTEDPORTS &
                   ea:HasElementaryAdaptations[HasPattern[{adaptationinst}]] &
                   ep1:HasEmittedPorts3[{HasGenericRoles(HasPattern(adaptationinst))(ApplyPosition(ea))}] &
                   ep2:HasAddedEmittedPorts[{adaptationinst}] &
                   HasPortToEmit2[{ep1}]*HasPortToEmit[{ep2}] <: FILTERS &
                   HasParticipants(adaptationinst)(HasTarget2(ep1)):HasTarget[{ep2}] |
              HasConcreteRoles[{HasParticipants(adaptationinst)(ApplyPosition(ea))}]*{ep2}) ||

        HasAdaptationPorts := HasAdaptationPorts -
             UNION ea.(ea:ELEMENTARYADAPTATIONS &
                       ea:HasElementaryAdaptations[HasPattern[{adaptationinst}]] |
             HasConcreteRoles[{HasParticipants(adaptationinst)(ApplyPosition(ea))}]*HasPointcut[{ea}])
    END
END
```

E
Acronymes

La signification des acronymes utilisés dans ce document est, en règle générale, précisée lors de leur première utilisation. Cette annexe regroupe tous ces acronymes, leur signification et une équivalence en français lorsque necessaire.

ADL Architecture Description Language, Langage de description d'architecture.

AOP Aspect Oriented Programming, Programmation orientée aspects.

API Application Programming Interface, Interface de programmation des applications.

ASSL A Snapshot Sequence Language, Langage de génération de configurations.

CCM CORBA Component Model, Modèle de composants CORBA.

CIDL Component Implementation Definition Language, Langage de définition des implantation de composants.

CORBA Common Object Request Broker and Architecture, Architecture commune pour bus d'invocation de requêtes d'objets.

COTS Component Off The Shelf, Composant sur étagère.

CTL Computational Tree Logic, Logique temporelle arborescente.

EJB Enterprise JavaBeans, Composants Java pour l'entreprise.

FAC Fractal Aspect Component, Composant d'aspect de Fractal.

IDL Interface Definition Language, Langage de définition d'interfaces.

IDM Ingénierie Dirigée par les Modèles (équivalent de l'acronyme anglais MDE).

IHM Interface Homme Machine.

ISL Interaction Specification Language, Langage de Spécification des Interactions.

ISO International Standardization Organization, Organisation internationale de standardisation.

JAC Java Aspect Components, Composants d'Aspect Java.

MDA Model driven Architecture, Architecture Dirigée par les Modèles.

MDE Model driven Engeneering (équivalent de l'acronyme français IDM).

MOF Meta Object Facility, Ressource pour les métaobjets.
MOP Meta Object Protocol, Protocole à métaobjets.
MolèNE MObiLE Networking Environement, Environnement pour réseaux mobiles.
OCL Object Constraint Language, Langage de contraintes des objets.
ODP Open Distributed Processing, Traitements informatiques ouverts distribués.
OMG Object Management Group.
OSGi Open Services Gateway initiative.
PDA Personal Digital Assistant.
PIM Platform Independent Model, Modèle indépendant des plates-formes.
PLTL Propositional Linear Temporal Logic, Logique temporelle linéaire propositionnelle.
PSM Platform Specific Model, Modèle spécifique aux plates-formes.
RMI Remote Method Invocation, Invocation de méthodes à distance.
SECRET Semantic Reasoning Tool, Outil de raisonnement sur la sémantique.
SOA Service Oriented Architecture, Architecture Orientée Service.
SoC Separation of Concerns, Séparation des préoccupations.
SOFA/DCUP SOFtware Appliances / Dynamic Component UPdating.
SOP Subject Oriented Programming, Programmation orientée sujets.
UML Unified Modeling Language, Langage de modélisation unifié.
USE UML based Specification Environment.

F
Publications

Ce chapitre regroupe toutes les publications produites pendant cette thèse en tant qu'auteur principal ou associé. Ces publications sont classées par rubrique et par ordre chronologique inverse à l'interieur des rubriques. Un certain nombre de ces publications sont disponibles à l'adresse : http : / / www.essi.fr / ~ occello.

Conférences internationales avec comité de sélection et actes

- A. Occello and A-M. Dery-Pinna. An adaptation-safe model for component platforms. In *13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04)*, Nice, France, 1-3 july 2004.

Atelier de travail international avec comité de sélection et actes

- A. Occello and A-M. Dery-Pinna. Safe runtime adaptations of components : a UML meta-model with OCL constraints. In *First International Workshop on Foundations of Unanticipated Software Evolution (FUSE'04)*, 28 march 2004.

Conférences nationales avec comité de sélection et actes

- A. Occello, A-M. Pinna-Dery. Sûreté de fonctionnement d'applications nomades construites par assemblage de composants. In *ACM UbiMob*, Grenoble, France, juin 2005.
- M. Bartorello, H. Maguin, A. Occello, M. Blay-Fornarino, A-M. Dery, Michel Riveill. Intégration de services au sein d'un serveur d'EJB. In *Langages et modèles à objets LMO'2002*, vol. 8 de L'Objet, Hermès, 2002, p.169-184.

Atelier de travail national avec comité de sélection

- A. Occello, A-M. Pinna-Dery. Projection d'un modèle de sûreté pour plates-formes à composants : retour d'expérience. In *Journée composants*, Le Croisic, France, avril 2005.
- A. Occello, A-M. Dery-Pinna. Approche semi-formelle pour l'adaptation dynamique de composants. In *Journées Objets, Composants et Modèles*, Berne, Suisse, mars 2005. GDR ALP.
- A. Occello, A-M. Dery-Pinna, M. Blay-Fornarino and M. Riveill. Contrôles des adaptations d'applications à base de composants. In *Journées Objets, Composants et Modèles*, Vannes, 5 février 2003. GDR ALP.

- A. Occello, M. Blay-Fornarino, A-M. Pinna-Dery, M. Riveill. Vers une adaptation dynamique cohérente des composants. In *Journée composants : Systèmes à composants adaptables et extensibles*, Grenoble, France, 17 et 18 octobre 2002.
- M. Blay-Fornarino, D. Ensellem, A. Occello, A-M. Pinna-Dery, M. Riveill, J. Fierstone, O. Nano, and G. Chabert. Un service d'interactions : principes et implémentation. In *Journée composants : Systèmes à composants adaptables et extensibles*, Grenoble, France, 17 et 18 octobre 2002.

Rapports de recherche et techniques

- F. Baude, E. Bruneton, T. Coupaye, Pierre-Charles David, Thomas Ledoux, Matthieu MOREL, A. Occello, M. Riveill, and A. Senart. Apports et limites de l'architecture ARCAD. Livrable d5.3, RNTL ARCAD, Laboratoire I3S, juin 2004.
- A. Occello and A-M. Dery-Pinna. Safety of component adaptations : Elements of formalization. Technical Report I3S / RR-2004-04-FR, Laboratoire I3S - Université de Nice-Sophia Antipolis, Bâtiment ESSI - BP145 - F-06903 Sophia Antipolis CEDEX, janvier 2004.
- F. Baude, T. Coupaye, E. Bruneton, D. Emsellem, O. Layaida, M. Morel, A. Occello, M. Riveill, and A. Senart. Document de mise en œuvre. Livrable d2.3, RNTL ARCAD, Laboratoire I3S, décembre 2003.
- A. Occello, A-M. Pinna-Dery, and M. Riveill. Outils d'adaptation. Livrable l7, RNTL ASPECT, Laboratoire I3S, septembre 2003.
- A. Occello, A-M. Pinna-Dery, and M. Riveill. Spécification de l'adaptation des composants. Livrable l6, RNTL ASPECT, Laboratoire I3S, septembre 2003.
- D. Emsellem, M. Blay-Fornarino, and Audrey Occello. Démonstration de test mettant en évidence différents protocoles d'interaction. Livrable d3.3, RNTL ARCAD, Laboratoire I3S, septembre 2002.

Résumé

Les technologies pour l'adaptation dynamique (composants, aspects, ...) arrivent à maturité et permettent de modifier les applications durant leur exécution. Si l'on considère que la sûreté de fonctionnement d'une application est la propriété permettant aux utilisateurs d'un système de placer une confiance justifiée dans le service qu'il leur délivre alors il faut garantir lors d'une adaptation dynamique que cette propriété est préservée. Autrement dit, une adaptation n'est pas « sûre » à partir du moment où le service que fournit l'application après adaptation diverge du service attendu par l'utilisateur.

Actuellement, il n'existe pas de solution appropriée au problème de la sûreté des adaptations dynamiques. En effet, un certain nombre de techniques (typage, model-checking, ...) destinées à concevoir et implémenter les systèmes informatiques de façon sûre peuvent être utilisées dans le cadre des adaptations statiques mais pas directement pour valider des adaptations dynamiques. De plus, la prise en charge de ces dernières doit tenir compte du risque qu'une adaptation se produise à un moment inadéquat dans l'exécution de l'application et implique de traiter les problèmes de sûreté parallèlement à l'exécution de l'application sans perturbation. Bien que les plates-formes permettant les adaptations dynamiques proposent des solutions, il n'existe pas de consensus autour des vérifications à effectuer dans un contexte dynamique. D'autre part, la mise en œuvre de ces vérifications reste souvent informelle ou à la charge du développeur d'applications.

Nous proposons d'identifier la sûreté d'une adaptation indépendamment des plates-formes, et de déterminer le moment où les modifications liées à une adaptation peuvent être prises en compte de façon sûre dans l'exécution de l'application. Cette approche est basée sur un modèle nommé Satin sur lequel des propriétés de sûreté sont exprimées et validées. Le modèle Satin est mis en œuvre sous la forme d'un service de sûreté que les plates-formes peuvent interroger pour déterminer si une adaptation donnée risque de briser la sûreté de fonctionnement de l'application.

Mots-clé Adaptations dynamiques, sûreté de fonctionnement, composants, aspects, ingénierie dirigée par les modèles.

Abstract

Dynamic adaptation technologies (components, aspects, ...) arrive at maturity and allow for applications to be modified during their execution. If we consider safety as the property that enable users to trust application behavior, then we have to ensure that this property is preserved by application adaptations during execution. Then, an adaptation is not safe as soon as the resulting application behavior does not meet anymore the user point of view behavior.

There are no appropriate solutions to handle safety for dynamic adaptation purpose. Some techniques (typing, model-checking, ...) that help designing and implementing computer systems safely can be used in the context of static adaptations but cannot be applied directly to dynamic adaptations. Indeed, managing dynamicity implies taking into account the risk that an adaptation may occur in circumstances that do not comply the application execution and implies recovering errors while the application runs. Platforms allowing for dynamic adaptation provide some solutions but there is not an unified approach of the problem. Moreover the implementations of the proposed solutions are often informal or left to the application developer instead of being offered by the platform.

Facing application evolutions, we propose to determine adaptation safeness independently of components platforms and aspect frameworks and to identify the circumstances under which it is safe to take into account a given adaptation in the application execution. Satin is a model describing the key elements involved in the adaptation process. Safety properties are expressed and validated over this model to identify the criteria of safe adaptations. Satin has been implemented as a safety service that platforms can query to determine whether a given adaptation can be performed without breaking the safeness of the overall application execution. A prototype of this service has been developed in Java so as to validate the applicability of the proposed approach.

Key words Dynamic adaptations, safety, components, aspects, model driven engineering.